SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS NONE
2a 2b 4. **AD-A218 377**	3. DISTRIBUTION/AVAILABILITY OF REPORT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.
	5. MONITORING ORGANIZATION REPORT NUMBER(S) AFIT/CI/CIA- 89-007
6a. NAME OF PERFORMING ORGANIZATION AFIT STUDENT AT Univ of Central Flordia 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION AFIT/CIA
6c. ADDRESS (City, State, and ZIP Code)	7b. ADDRESS (City, State, and ZIP Code) Wright-Patterson AFB OH 45433-6583
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification) (UNCLASSIFIED)
A Relational Object-Oriented Management System and An Encapsulated Object Programming System

12. PERSONAL AUTHOR(S)
Michael L. Nelson

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
~~THESIS~~/DISSERTATION	FROM _____ TO _____	1988	256

16. SUPPLEMENTARY NOTATION
APPROVED FOR PUBLIC RELEASE IAW AFR 190-1
ERNEST A. HAYGOOD, 1st Lt, USAF
Executive Officer, Civilian Institution Programs

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

90 02 14 047

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL ERNEST A. HAYGOOD, 1st Lt, USAF	22b. TELEPHONE (Include Area Code) (513) 255-2259 22c. OFFICE SYMBOL AFIT/CI

DD Form 1473, JUN 86 *Previous editions are obsolete.* SECURITY CLASSIFICATION OF THIS PAGE

AFIT/CI "OVERPRINT"

A RELATIONAL OBJECT-ORIENTED MANAGEMENT SYSTEM AND AN ENCAPSULATED OBJECT-ORIENTED PROGRAMMING SYSTEM

by

MICHAEL L. NELSON

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science at the University of Central Florida Orlando, Florida

December 1988

Major Professor: Ali Orooji

ABSTRACT

The purpose of the Relational Object-Oriented Management System (ROOMS) is to show that the relational database scheme is a viable approach for storing object-oriented data. ROOMS is designed so that it can be implemented in any object-oriented language with appropriate I/O commands, or added to any object-oriented database management system that allows user-defined collections of data.

Various problems were encountered in developing ROOMS. While these problems have been solved, the best solution is to use the Encapsulated Object-Oriented Programming System (EOOPS). EOOPS is based upon an inheritance scheme which preserves encapsulation. This encapsulated approach avoids the problems associated with the name conflicts that occur with "conventional" object-oriented languages. EOOPS also includes a proper metaclass and allows for generic routines.

ROOMS was then reimplemented in EOOPS to study the enhancements provided by EOOPS. As expected, the encapsulated form of inheritance provided in EOOPS was responsible for most of these enhancements. It led to a simplified record structure which in turn led to a simplified implementation of the relational operations.

ACKNOWLEDGEMENTS

Special thanks to Dr. Orooji, the chairman of my Research Committee. I cannot begin to count the number of times that he went out of his way to lend a helping hand.

Thanks also to the rest of my Research Committee - Dr. Moshell (who was the first one to get me interested in object-oriented programming), Dr. Hughes (who was a great help in making comparisons to Smalltalk), Dr. Myler (who gave me the idea for a wonderful example of a potential user of my database system), and Dr. Cottrell (who was especially helpful in getting us through all of the paperwork quirks, but unfortunately could not be here the semester that I graduated).

I would also like to thank my wife Debbie for putting up with me and for helping me through all the really tough times. And little Mikey is the one who helped me to keep all my priorities straight - one look at that little guy was all it took to remember what the most important things really were, and that everything would work out all right in the end.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

ADT - Abstract Data Type

AI - Artificial Intelligence

CAD - Computer-aided Design

CAE - Computer-aided Engineering

CAM - Computer-aided Manufacturing

CASE - Computer-aided Software Engineering

CLOS - Common Lisp Object System

CWA - Closed World Assumption

DBS - Database System

DBMS - Database Management System

EOOPS - Encapsulated Object-Oriented Programming System

EOOPS/ROOMS - EOOPS version of ROOMS

I/O - Input/Output

OOP - Object-Oriented Programming

OOPS - Object-Oriented Programming Systems

OOPSLA - Object-Oriented Programming Systems, Languages, and Applications

PCS - PC Scheme

PCS/ROOMS - PCS version of ROOMS

ROOMS - Relational Object-Oriented Management System

SQL - SEQUEL, a relational DBMS query language

TI - Texas Instruments

INTRODUCTION

A database has been defined as a collection of interrelated data stored together with controlled redundancy to serve one or more applications in an optimal fashion (Martin 1976). The data is stored independent of the programs which use the data, and a common and controlled approach is used in adding new data and in modifying and retrieving existing data (Martin 1976).

Various data models have been used in existing commercial database systems, with the three most important being the hierarchical, network, and relational data models (Ullman 1982). However, conventional database systems that are based upon these models have various limitations on the data types and operations that they support (Andrews 1987; Caruso 1987; Copeland 1984; Maier 1986; Ong 1984; Osborn 1986), and do not address the needs of many non-traditional applications such as cartography and geographic systems and computer-aided design (Orenstein 1986).

Like many new ideas, object-oriented programming (OOP) does not yet have a universally accepted definition. However, the following definition

(equation) (Wegner 1987) seems to be what (most of) the user community agrees on:

object-oriented = objects + classes + inheritance

where a class can be thought of as the definition of an abstract data type and the operations that can be performed on it, an object is an instance of a class, and inheritance is a mechanism that is used for sharing common features between classes.

The Relational Object-Oriented Management System (ROOMS), designed and implemented as part of this research project, shows that it is possible for the user to treat objects no differently from conventional data. In other words, ROOMS removes the data type limitations that exist in conventional database systems, and allows object-oriented users to store objects in the database utilizing a relational scheme.

In ROOMS, a database is simply a collection of objects called relations. Similarly, a relation is a collection of objects called records. Further, a record consists of various fields, which are user-defined objects. No distinction is made between simple objects (such as conventional data) and more complex objects (such as pictures or digitized voice).

In designing and implementing ROOMS, several problems (not necessarily unique to ROOMS) were encountered due to the way that various OOP features

have been implemented in existing languages. Therefore, an object-oriented language called Encapsulated Object-Oriented Programming System (EOOPS) was designed to correct these deficiencies. Note that EOOPS is not intended to be a complete language. Only those features unique to an OOP language were included. Thus, EOOPS might be considered a "bolted-on" OOP package that could be added to any conventional language, along the lines of SCOOPS, the object-oriented part of PC Scheme (Texas Instruments Inc. 1987a,b), or it could serve as the core of a language in which OOP principles are "built-in." Rather than a formal language specification, a language reference manual which describes the various features of EOOPS was developed. An actual implementation of EOOPS was considered to be beyond the scope of this research effort.

ROOMS was then "reimplemented" in EOOPS to show that the various problems have been solved in ways that result in a cleaner, easier to understand program. Since EOOPS itself was not actually implemented, this "implementation" of ROOMS in EOOPS took the form of a high-level (pseudocode) program.

Chapter 1 presents a survey of the literature to provide a context for the research effort. An overview of object-oriented programming, relational database systems, and object-oriented database systems provides a

common starting point for the development of ROOMS and EOOPS. ROOMS is described in detail in Chapter 2, along with a discussion of the problems encountered in developing that system. Chapter 3 (and Appendix A) describes EOOPS in detail. The implementations of ROOMS in PC Scheme and in EOOPS, along with a comparison of these implementations, are described in Chapter 4 (and Appendices B and C). Finally, in Chapter 5 the research effort is summarized, and suggestions for future research are presented.

CHAPTER 1

SURVEY OF THE LITERATURE

In this chapter, object-oriented programming (OOP) will be explored and discussed. Since terminology varies from one OOP language to another, the various terms and definitions will be presented, and the terminology as used in this project is outlined. Genericity, which can be compared with inheritance as it exists in OOP (Meyer 1986; Touati 1987), will also be covered. Finally, relational database systems and object-oriented database systems will be examined, both in general terms and various specific implementations.

Object-Oriented Programming Systems

As mentioned in the introduction, OOP does not yet have an exact definition that we can call upon. Some studies (Rentsch 1982; Snyder 1986b) attempt to clarify OOP by characterizing its more important aspects rather than giving a definition. The following quote from Rentsch (1982, 51) is a wonderful tongue-in-cheek way of answering the question: 'What is object-oriented programming?'

> My guess is that object-oriented programming will be in the 1980's what structured programming was in the 1970's. Everyone will be in favor of it. Every manufacturer will promote his products as supporting it. Every manager will pay lip service to it. Every programmer will practice it (differently). And no one will know just what it is.

This quote from Cox (1986, 29) expresses a similar idea:

> Object-oriented is well on its way to becoming the buzzword of the 1980's. Suddenly everybody is using it, but with such a range of radically different meanings that no one seems to know exactly what the other is saying.

And the following quote from Stroustrup (1988, 10) simply states that object-oriented programming is "good":

> Object-oriented has become a buzzword that implies "good" programming.

So, how do we define OOP? It has been said that the OOP concepts of messages/objects replaces the concepts of operators/operands from conventional languages (Cox 1984). OOP can also be viewed as an abstraction mechanism that is a technique for organizing very large programs (Moon 1986). And OOP can be defined in terms of its fundamental concepts - types, type hierarchies, and inheritance (Halbert 1987) (the term "type" is used in some OOP languages instead of "class").

OOP has also been defined using the following equation (Wegner 1986, 177):

object-oriented = data abstractions + abstract data types + type inheritance

This has been modified slightly to (Wegner 1987, 169):

object-oriented = objects + classes + inheritance

This is the definition that we prefer, and it seems to be the definition accepted by most of the attendees at the OOPSLA'87 conference that was held in October 1987 in Orlando, Florida.

Smalltalk (Digitalk 1986) is widely recognized as one of the earliest object-oriented languages. Alan Kay (one of the developers of Smalltalk) is credited by Rentsch (1982, 56) as saying that "...the entire thrust of Smalltalk's design has been to supersede the concept of data and procedures entirely; to replace these with the more generally useful notions of activity, communication, and inheritance."

Since it is difficult if not impossible to come up with a succinct definition of OOP, let us study the more common features of OOP - objects, classes, messages, metaclasses, inheritance, delegation, and encapsulation. Composite objects, objects which consist of other objects, will also be discussed.

Objects, Classes, Messages, and Metaclasses

One of the central themes of object-oriented programming is that every entity is an object (Briot 1987). An object is an instance of a class, and it

responds to messages. But just how do we define objects, classes, and messages?

Objects can be viewed in many ways. An object can be said to represent entities and concepts from the application domain being modeled (Fishman 1987). Some studies (Cox 1984; Stroustrup 1986b) define objects as private data or regions of storage. Others (Meyer 1987b; Texas Instruments Inc. 1987b) define an object as an instance of an abstract data type or an abstract entity. Other papers (Cox 1986; Diederich 1987; Halbert 1987; Schmucker 1987; Stefik 1986; Wegner 1987; Wiederhold 1986) go a step further and define an object as having its own data or local memory (called instance variables) that recognizes a set of procedures (called methods, which are invoked by sending messages to the object) for manipulating its local memory.

The TI Scheme Language Reference Manual (Texas Instruments Inc. 1987b) contains a "circular" definition of an object - an object is an instance of a class, and a class contains the description of one or more similar objects.

As previously mentioned, the terms "type" and "class" are generally used to mean the same thing. C++ (Stroustrup 1986b), however, is an exception in that it defines a "type" as a system supplied data type and a "class" as a user-defined "type." We will take the more

commonly held view that the terms have the same meaning. The term "class" is used in all of the languages (C++, PC Scheme, and Smalltalk) that we employed for this research project, and therefore it is the term that will be used in this dissertation. Actually, we would prefer the term "type" - somehow it just seems more "natural" to say that an object is of type X rather than saying that an object is an instance of class X. Also, the term "type" helps to convey the idea that it is somehow an extension of the idea of abstract data types.

"Classification arises from the universal need, in any domain of discourse, to describe uniformities of collections of instances" (Wegner 1986, 173). It has also been said that objects are classified by type, with objects sharing common properties belonging to the same type (Fishman 1987). A class can be defined as the shared portion of similar objects (Schmucker 1987). Similarly, a class can be thought of as a description of similar objects, sort of an abstract data type definition (Meyer 1987a; Stefik 1986; Wegner 1987). A class can also be thought of as a template or cookie cutter from which objects can be created (Wegner 1987).

A class declaration typically includes the class name and definitions for the class variables, instance variables, and methods (Snyder 1986b; Texas Instruments Inc. 1987b). The class name is simply the name of the

class, a way to refer to it and identify it. Class variables contain information that is shared by all instances of the class, i.e., all instances of a class share the class variables by both name and value. Instance variables are local to each instance of a class and contain information specific to that instance. That is, each instance of a class has its own set of instance variables. The names of the variables in this set are identical from one object to another, but the values may vary.

Methods are operations or procedures which determine the behavior of instances of a class, i.e., the methods represent the only operations that an object can be told to perform.

A message is sent to an object to request that it perform one of its operations (Cox 1984; Schmucker 1987; Seidewitz 1987). These operations are said to be polymorphic (Cardelli 1985; Meyer 1987a; Stefik 1986; Stroustrup 1986b) in that different classes can have operations with the same name. This is similar to operator overloading, which exists in some languages (such as Ada and C) (Booch 1987; Cardelli 1985; Stroustrup 1986a; Touati 1987). The difference is that in conventional languages, even with overloading, the environment must tell how each command should be performed by naming a specific piece of executable code

appropriate to the data type at hand, while in a message/object environment, one tells the object what to do and the object uses its methods to decide how to do it (Cox 1984).

We have now discussed objects, classes, and messages, but one point has not been covered - where do these objects come from? And, if every entity in an object-oriented language is supposed to be an object, how do we explain the existence of a class which does not appear to be an object?

The concept of a metaclass (Bobrow 1986; Briot 1987; Cointe 1987; Cox 1986; Digitalk Inc. 1986; Stefik 1986) is used in some languages such as CommonLoops (Bobrow 1986), the Common Lisp Object System or CLOS (Bobrow 1988), and Smalltalk (Digitalk Inc. 1986). Classes are simply defined to be objects themselves, with a class being an instance of a metaclass. A "create" message can now be sent to the class (which is an object), so that messages are still sent only to objects. The metaclass itself is a special object provided by the programming language which responds to messages to create new classes.

CLOS (Bobrow 1988) allows multiple metaclasses, including user-defined metaclasses. The metaclass which is used determines the representation of instances of its instances (i.e., the instances of the classes which

the metaclass is used to create), the form of inheritance used by the classes which the metaclass is used to create, and the particular forms of optimization which are used. A default metaclass, called standard-class, is provided which "is appropriate for most programs" (Bobrow 1988, 1-9).

ObjVlisp (Cointe 1988), a system used to simulate object-oriented language extensions, also allows multiple metaclasses and user-defined metaclasses. However, allowing user-defined metaclasses brings about the possibility of an "infinite regress ... (where) ... a metaclass is a class which instantiates a class, a metametaclass is a class which instantiates a metaclass, a metametametaclass ..." (Cointe 1988, 160). Therefore, in ObjVlisp a special class called Class is defined to create metaclasses, and it is defined as an instance of itself.

Other languages, such as Objective-C (Cox 1986), define classes to be concepts rather than objects, and provide factory objects which are used to create instances of a class. A factory object is automatically built by the programming language for each class that has been defined. Messages are then sent to factory objects to create new instances of the class (Cox 1986; Schmucker 1987), so messages are still sent only to objects.

Still other languages, such as C++ (Stroustrup 1986b) and PC Scheme (Texas Instruments Inc. 1987a,b), "ignore" the problem altogether. C++ simply says that a class can have "constructor" and "destructor" functions to create and delete objects. Similarly, PC Scheme provides a "make-instance" procedure to create instances of a class.

It is our opinion, however, that all of these approaches can be considered to be equivalent to the metaclass concept. All the languages that we have investigated have some type of a "define-class" operation and some type of a "create-instance" operation. The "define-class" operation can simply be thought of as a message that is sent to the metaclass to create an object which is a class. The "create-instance" operation then becomes a message which is sent to the class object.

Inheritance and Delegation

A class can inherit the properties or characteristics of another class (Borning 1982; Halbert 1987; Schaffert 1986; Stein 1987; Stroustrup 1986b), i.e., it inherits the class variables, instance variables, and methods of the other class (Cox 1986; Snyder 1986b; Stefik 1986; Texas Instruments Inc. 1987b; Thomas 1987; Wegner 1987). Inheritance can also be

thought of as a form of code sharing (Sandberg 1986; Thomas 1987) or resource sharing (Wegner 1987). It has been said that inheritance is the one feature of OOP languages that really sets them apart from other languages (Thomas 1987).

The inheritance relationship between classes is referred to by several different names - superclass/subclass, supertype/subtype, parent/child, and ancestor/descendant. A set of classes related by inheritance is referred to as a class hierarchy or an inheritance hierarchy (Cox 1986; Sandberg 1986; Stefik 1986; Stroustrup 1986b). We will use the terms superclass/subclass to indicate one level up or down the hierarchy, and the terms ancestor/descendant to indicate one or more levels.

Some OOP languages allow inheritance from only one superclass. This is referred to as single inheritance or simply as inheritance. Other OOP languages allow inheritance from more than one superclass. This is called multiple inheritance.

Note that a potential problem exists when inheritance is allowed - name conflicts. With both single and multiple inheritance, a newly defined variable or method may have the same name as an inherited variable or method (either by choice or by accident). With multiple inheritance, a class may

inherit variables or methods with the same name from different ancestors - a decision must be made as to whether we attempt to make some distinction between them, combine them together, or somehow make a choice among them. In Chapter 2 we will discuss how different OOP languages handle these issues, and in Chapter 3 we will show how we handle them in EOOPS. The set of classes related by multiple inheritance can still be referred to as a hierarchy (Hendler 1986), or it may be referred to as a lattice (Stefik 1986).

Delegation also allows the sharing of both variables and methods. Inheritance, however, refers to sharing between classes while delegation is concerned with sharing between individual objects (Stein 1987). Inheritance can be thought of as implementing sets (classes), and delegation can be thought of as implementing prototypes (Lieberman 1986). Inheritance and delegation can be considered alternate methods for incremental definition and sharing (Stein 1987). Inheritance can be simulated using delegation, but delegation cannot be fully simulated using inheritance (Lieberman 1986; Stein 1987).

A delegation-based language can be defined as an object-based language that supports classless objects and delegation, i.e., "delegation-based = objects - classes + delegation" (Wegner 1987, 173). A delegation

hierarchy is the set of relationships between various objects (Stein 1987).

The ancestor object, called a prototype, is both an object in its own right and a template for its descendants - it provides default values and operations to its descendants (Wegner 1987). An object is said to delegate an attribute (method or variable) to its ancestor (a prototype) if it does not maintain a value for that attribute (Stein 1987).

The terms delegation and prototype are often used only in connection with one another, and in comparison with inheritance. Delegation, however, can be defined more simply as a technique for forwarding a message off to be handled by another object (Stefik 1986). Using this definition, we believe that inheritance could be viewed as a specific form of delegation, with messages being delegated to an object's class.

Abstract Classes and Standard Protocols

An abstract class is a class which has no instances, and it usually has no variables (Johnson 1988; Pinson 1988). Its purpose is to provide a standard protocol, sometimes referred to as a common protocol or simply as a protocol, to its subclasses. This standard protocol is a set of undefined methods which must be implemented by the subclasses (i.e., only

the name of the method is given in the abstract class) (Pinson 1988). This can also be thought of as a program skeleton (Johnson 1988), where the user is responsible for filling in certain options, and is sometimes referred to as subclass responsibility.

A class that is not abstract (i.e., a class with instances) is said to be concrete (Johnson 1988). It should be noted that the abstract class is a "concept" and is not implemented as a construct in any of the object-oriented languages that we have studied. There is nothing in the definition of an abstract class to differentiate it from a concrete class, and there is nothing in its implementation to prevent it from having instances (and thus becoming a concrete class).

The idea of a protocol is normally associated with Smalltalk (Digitalk Inc. 1986) systems, although this concept could be implemented in other object-oriented languages. C++ (Stroustrup 1986b), for example, provides for virtual methods (called virtual functions in C++ terminology) which specify a method name that must be implemented by its subclasses. Even in a language which provides no support for this concept, methods could be written which expect other methods to be provided by its subclasses.

Encapsulation

Encapsulation can be thought of as a form of information hiding (Diederich 1987), or as the strict enforcement of information hiding (Micallef 1988). It has been said to be the foundation of the whole approach to object-oriented programming (Cox 1986). Encapsulation is supported in OOP languages by allowing an object to be manipulated only by the set of operations defined on it (Snyder 1986b). This set of operations is called the external interface of the object. This external interface can be thought of as a "wall" of code around each piece of data (i.e., around each object) which restricts the effects of change (Cox 1986). Encapsulation is said to be enforced since sending a message to an object is the only way to modify its variables (Cox 1986; Diederich 1987; Snyder 1986b).

However, inheritance can severely compromise encapsulation (Snyder 1986b). A module (such as a class definition) is said to be encapsulated if its clients (users) can only access it via its defined external interface (Snyder 1986b). Thus from an end-user's (i.e., non-developer's) point of view, classes and objects in an OOP language are encapsulated. Inheritance, however, introduces another type of client of a class - a subclass. The subclass has access to all of its inherited variables by name, and is not

restricted to the external interface of its ancestor class (Snyder 1986b). Thus, encapsulation has been violated by inheritance.

This anomaly/contradiction is not even mentioned in most of the OOP literature that we have reviewed. Most papers simply discuss encapsulation and inheritance with no mention of the effects that inheritance can have on encapsulation - it has even been said that encapsulation and inheritance are two of the most important aspects of object definition (Cox 1986; Diederich 1987). The most notable exception is one paper (Snyder 1986b) that is entirely devoted to the subject. Another exception (Cox 1986) is the idea that implementors have both inheritance and encapsulation available to them. Inheritance is claimed to be an implementation issue, not a design issue. While it is never stated by Cox (1986) that inheritance can violate encapsulation, it is said that declaring the type of an instance variable to be another class is a form of inheritance and encapsulation combined. A fairly recent paper (Micallef 1988) also discusses these issues and compares the level of encapsulation provided by various object-oriented languages.

Composite Objects

A composite object can be defined as an object that consists of other objects, i.e., the variables of a composite object are defined as being objects themselves (called dependent objects) (Kim 1987; Stefik 1986). A composite object is also referred to as an aggregate object (Halbert 1987). It is interesting to note that none of these papers make any mention of using composite objects as a way of preserving encapsulation, while Cox (1986), as mentioned above, says that this is a form of encapsulation without giving the concept a name (and with only a very brief discussion of the subject).

From a database point of view, composite objects can be used to define a unit of storage and retrieval to enhance performance of the system (Kim 1987). Retrieving (storing) an object implies retrieving (storing) all of its dependent objects and as such helps to resolve the problem of what data should be resident in memory at a given time.

Composite objects can also be viewed as an alternative to multiple inheritance, choosing one or the other depending on what the relationship between the classes should be (Halbert 1987). Multiple inheritance is said to be appropriate when an object is the sum of the behavior of its parts; composite objects, on the

other hand, are appropriate when an object is more simply the sum of its parts (Halbert 1987).

For example, a class Airplane could be built using multiple inheritance from the superclasses Wings, Fuselage, Engine, and Tail. However, this would be a conceptual error since an airplane is composed of these parts - it is not the sum of their behaviors. As such the class Airplane should be a composite object with instance variables consisting of these other classes (Halbert 1987).

As an alternative example, consider a class Teaching-Assistant, to be constructed using the classes Teacher and Student. Using multiple inheritance in this case is appropriate as a Teaching-Assistant should be the sum of the behaviors of a Teacher and a Student. Using a composite object consisting of a Teacher and a Student would be inappropriate as a Teaching-Assistant does not consist of a Teacher and a Student (Halbert 1987).

We have been using another form of a composite object which, to our knowledge, has not been discussed in the literature. Rather than using dependent objects, we are using subobjects. With a subobject, the value of an instance variable is a pointer to another object. A subobject is in no way dependent upon its composite object; it exists as a separate entity.

The comparison of subobjects as we have been using them with dependent objects as discussed in the literature is itself an interesting subject. A dependent object is by definition completely dependent upon its composite object - it cannot be created, accessed, or deleted without doing so through the composite object. And if the composite object is deleted, all of its dependent objects are also deleted (if not, there would not be any way to access them). A subobject, on the other hand, can be created, accessed, or deleted directly (without even knowing that it is a subobject), or through its composite object. If a composite object consisting of subobjects is deleted, a decision must be made as to whether or not to delete its subobjects, although a garbage collection system could delete unused subobjects from memory after the composite object has been deleted. Another difference is that dependent objects cannot be shared; subobjects do not have this restriction.

We are tempted at this point in time to say that subobjects are "more powerful" than dependent objects. Subobjects can be restricted in usage to (in effect) implement dependent objects, but since dependent objects cannot exist as standalone objects, the reverse is not true. However, with more "power" comes more (potential) problems. Being able to modify or delete a subobject

without the composite object's knowledge could definitely lead to various problems such as inconsistencies and "dangling pointers."

Genericity

Genericity is a mechanism for building reusable software components (Booch 1987; Touati 1987). A generic definition does not in itself create any code; it only defines what might be thought of as a template for a program unit. The user must instantiate a generic unit to create an instance of the generic code (Booch 1987). Popular examples of generic routines are a swap routine to exchange the values of two variables and a sort routine to order a set of variables by their values (Booch 1987; Meyer 1986). Rather than defining a swap (or sort) routine for every data type for which it is needed, a generic version is written and then instantiated for each data type.

Most discussions concerning both OOP and genericity are only concerned with comparing them (Meyer 1986; Seidewitz 1987; Touati 1987). Both genericity and inheritance apply some form of polymorphism (the ability to define program entities that may take more than one form) (Meyer 1986). It can be shown (Meyer 1986) that genericity can be simulated using inheritance, but that inheritance cannot be

simulated using genericity. As such, inheritance is said to be a more powerful mechanism than genericity. However, the simulation of genericity using inheritance is very verbose and overly complex (Meyer 1986).

Eiffel (Meyer 1986; Meyer 1987; Meyer 1988) is an object-oriented language which offers both inheritance and a limited form of genericity. Classes may have generic parameters representing types. For example, ARRAY [T] is a system defined class with a generic parameter. When instances of the class ARRAY are instantiated, the parameter T may be set to any other previously defined class. This provides a convenient way of defining arrays of integers and arrays of characters using the same definition for the class ARRAY.

C++ (Stroustrup 1986b) provides a similar construct in which a generic collection, such as a list, can be defined and then instantiated to contain any other class of objects. CLOS (Bobrow 1988) provides generic functions, but these are apparently equivalent to virtual functions in C++, in that they are the way that CLOS handles operator overloading (i.e., functions or methods with the same name).

We would like to see some form of genericity included in all object-oriented languages, preferably one that is at least as powerful as the idea of generic

collections. It would be very helpful in the development of complex applications such as ROOMS.

Database Management Systems

A database (sometimes written data base) is a collection of data stored "permanently" in a computer (Hutt 1978; Martin 1976; Ullman 1982). Objects (data) to be stored in a database are said to be persistent in that they survive beyond a single programming session (Andrews 1987; Buneman 1986; Caruso 1987; Fishman 1987; Thatte 1987; Wiederhold 1986). A database management system (DBMS) or database system (DBS) is the software that allows user(s) to use and maintain the data in the database (Hutt 1978; Ullman 1982).

The database architecture or data model refers to how the data is organized in the database and how it appears to the user (Brackett 1987; Ullman 1982). The three most important conventional data models (Bic 1986; Brackett 1987; Ullman 1982) are the relational model, the network model, and the hierarchical model. It can be shown that all three models are equivalent in modeling power (Ullman 1982), with relational databases being the easiest to design, maintain, and use (Bic 1986; Brackett 1987). The simplicity of the relational model has made it the most widely used data model.

We will now explore the concepts of relational database management systems in more detail. This will be followed by a discussion of object-oriented database systems. We will also discuss why conventional database systems do not seem to be appropriate for object-oriented applications.

Relational Database Systems

The relational data model is (essentially) based on the concept of flat files (or tables or arrays), with each table called a relation (Bic 1986; Codd 1970; Gallaire 1984; Martin 1976; Ullman 1982). A tuple of a relation is simply a row in the table. The columns of a relation are called attributes. The arity of a relation is the number of columns that it has.

From a database point of view, relations must be finite. This eliminates operations such as complement, as -R (the set of all tuples not in R) would denote an infinite relation (Ullman 1982). This leads to what is usually called the closed world assumption (CWA), also called the convention for negative information (Gallaire 1984; Reiter 1978) - if r cannot be proved to be a member of R, then r is assumed to be in -R. Two other assumptions (Gallaire 1984) are the unique name assumption, which states that individuals with different names are different, and the domain closure assumption,

which states that there are no other individuals than those in the database. These assumptions allow queries such as "all employees not in the toy department" to be answered.

Query languages for the relational model can be divided into two main categories (Ullman 1982): those based on relational algebra, and those based on relational (predicate) calculus. The calculus languages can be further divided into tuple relational calculus and domain relational calculus. It can be shown (Ullman 1982) that relational algebra and relational calculus (both tuple and domain) are equivalent. Therefore, we will concentrate on relational algebra, which is more widely used and chosen to be implemented in ROOMS.

There are five basic operations that define relational algebra; all other operations can be built upon these five operations (Ullman 1982). They are as follows (where R and S are relations):

(1) Union: R U S, is the set of tuples that are in R or S or both; union is only applied to relations with the same arity.

(2) Set difference: R - S, is the set of all tuples in R but not in S; set difference is also only applied to relations with the same arity.

(3) Cartesian product: R X S, is the set of tuples rs (i.e., r concatenated with s) for every r in R and every s in S.

(4) Projection: PROJECT (x,y,...,z) R (where x,y,...,z is a subset of the attributes (columns) of R), is the set of tuples containing only the attributes x,y,...,z of the tuples in R.

(5) Selection: SELECT (F) R (where F is a formula involving (a) operands that are attributes or constants, (b) the arithmetic comparison operators <, =, >, ~=, >=, and <=, and (c) the logical operators AND, OR, and NOT), is the set of tuples in R that satisfy F.

These five basic operations have been implemented in ROOMS.

Existing commercial (relational) database systems lack various features that are needed in an object-oriented database (Copeland 1984). Perhaps the biggest limitation is that they generally offer only a fixed set of data types (such as integers and character strings), and do not have the capability to define even simple new types and operations, much less complex types and an inheritance scheme (Andrews 1987; Bloom 1987; Caruso 1987; Copeland 1984; Maier 1986; Smith 1987). The record structuring capabilities are inadequate in that every record of a given type must be identical in structure to all other records of that type (Copeland 1984). Null values may be available for missing fields, but fields that do not fit the "norm" must be somehow forced to fit the given structure. This leads to a limitation in modeling power in that much of the real world data gets over-simplified in the database (Copeland 1984).

It is interesting to note that we actually use abstract data types in both conventional database systems and conventional languages without realizing it (Ong 1984). The only truly non-abstract type is the bit string. All other types have an internal representation in the computer, a string of bits, and an external representation, a string of characters or digits. The problem is in the limited number of types that are available in conventional systems and the difficulties encountered in attempting to add user-defined abstract data types.

One problem with attempting to store abstract data types in a conventional database is that the object must be "flattened out" to fit the existing types. This usually results in loss of abstractness (and encapsulation) in that the user can now see the structure of the object. Two exceptions are INGRES-ADT (INGRES - Abstract Data Type) (Ong 1984) and RAD (Osborn 1986). In INGRES-ADT, abstract data type facilities have been added to INGRES, an existing relational database system. RAD, on the other hand, is an experimental relational database system which was designed to support abstract data types.

In order to use an abstract data type in INGRES-ADT (Ong 1984), the new type and its operations must be registered with the database manager. This registration

must include an internal representation (how the type is to be stored) and an external representation (how the type is to be displayed). Conversion routines (from internal to external form and from external to internal form) must also be registered. Operations on the type, along with the precedence of the operators must also be registered (this facility also allows new operations to be defined on existing data types). Two areas of possible future extensions to INGRES-ADT have been proposed - query optimization within INGRES-ADT and allowing for an abstract data type inheritance hierarchy (Ong 1984); as far as we know, neither of these extensions has been accomplished. However, a new object-oriented database system called POSTGRES (Stonebraker 1986) is being developed as a successor to INGRES - this system will be discussed further in the section on Object-Oriented Database Systems in this chapter.

RAD (Osborn 1986) was developed at about the same time as INGRES-ADT and is similar to INGRES-ADT in that it allows new data types and their operations to be added to the database. However, RAD does not allow for new operations to be added to existing data types.

Object-Oriented Database Systems

Object-oriented languages generally lack support for persistent objects, but conventional database systems often lack the expressibility of object-oriented languages (Andrews 1987; Bloom 1987; Caruso 1987; Copeland 1984; Merrow 1987). An object-oriented database management system (OODBMS) is able to offer support for persistent objects by providing a storage management facility which includes the features and expressibility of an object-oriented programming system.

Object-oriented languages are being extended in the direction of databases, and conventional database systems are being extended with object-oriented ideas (Bloom 1987; Thatte 1987). However, there is some concern that these two approaches are not as compatible as they may appear on the surface (Bloom 1987). Perhaps a more integrated approach is necessary, an object-oriented database programming language (Andrews 1987; Bloom 1987; Caruso 1987; Copeland 1984). This should help to simplify the programming process as the programmer does not have to be aware of two distinct systems (a database system and a programming language) (Bloom 1987). An integrated approach should also lead to improved performance as storage management can be optimized for a single language (Bloom 1987).

Regardless of the approach taken, however, the general idea is to merge database technology with the object model (Merrow 1987).

It is also important to provide interfaces to "conventional" object-oriented languages (Caruso 1987; Fishman 1987; Maier 1986). This, however, can lead to an "impedance mismatch" (Copeland 1984). One such mismatch is conceptual, in which the database language and the programming language support different programming paradigms, e.g., one could be a declarative language while the other is procedural (Copeland 1984). Another such mismatch is structural, in which the database language and the programming language do not support the same data types (Bloom 1987; Copeland 1984).

Conventional (relational) database systems generally provide "set-at-a-time" access to the data, i.e., one accesses a relation (set) and then the records in the relation. Many object-oriented applications (such as engineering) need "object-at-a-time" access, for access to individual objects that are not necessarily a member of any particular set. The following ideas on the subject are discussed by Thatte (1987). What we probably need in an object-oriented database system is both "object-at-a-time" and "set-at-a-time" access to the data. Historically, people have tried to use a single database system for all of their

applications. Most companies are not interested in having two incompatible database systems, one for engineering applications and another one for more conventional applications. They would prefer to have a single database system that can handle all of their data and all of their applications.

We will now discuss several object-oriented database management systems, including both commercially available systems and research systems. We will then briefly summarize the current status of the field.

Analyst. Analyst (Conrad 1987; Xerox 1987; Xerox 1988) is a product of the Xerox Corporation. It is quite different from the other systems that will be discussed, as it is not touted as an object-oriented database system. Rather, it is called an information analysis tool (Xerox 1987) or a large scale, multi-media data storage, retrieval, and manipulation system (Conrad 1987).

The primary goal of the Analyst was to "create a system readily accessible to the average (non-computer oriented) user, providing a personal, integrated, and extensible environment employing a uniform graphical interface" (Xerox 1987, 2). The Analyst is a single user system that was developed in Smalltalk-80.

The Analyst organizes data into information centers (likened to a file cabinet) which are further organized into smaller collections called folders. It uses a network filing structure (i.e., a navigational approach where a user "travels" through the system from one object to the next). It also includes a menu driven query system.

Document processing is supported, and graphical tools are provided to handle images, maps, and charts. These tools include the ability to "zoom in" on specific parts of a map or image. A spreadsheet is also included, which is said to be able to handle any arbitrary object type in a cell.

GemStone. GemStone (Caruso 1987; Copeland 1984; Maier 1986; Penney 1987; Servio Logic Corp. 1988a,b) is a product of Servio Logic Corporation. It aims at several application domains that are not served very well (if at all) by conventional database systems. These application domains include computer-aided design and engineering (CAD and CAE), computer-aided software engineering (CASE), artificial intelligence (AI), cartography, and electronic publishing.

GemStone was originally designed to use IBM PC workstations connected to a VAX computer. It now runs on VAX and Sun computers, and works with IBM PC's, Apple

Macintosh II's, Sun, or Tektronix workstations. Servio Logic Corporation is currently working to add additional computers to these lists.

OPAL, the database language used in GemStone, was designed as an enhancement to Smalltalk-80. OPAL allows for transaction control among multiple users (Smalltalk-80 is a single-user system), handles both larger objects and more objects (up to 2 billion) than Smalltalk-80 does, and includes declarative constructs for data manipulation (Smalltalk-80 is a strictly procedural language). It also supports object histories (i.e., changes to an object over time), allows for optional variables (with no storage penalty for objects which do not include the optional variables), and has the ability to add new variables to existing objects when a class definition is modified. Interfaces (that are somewhat more limited than OPAL) have been added for Smalltalk, C, C++, Objective-C, Fortran, Pascal, and Ada.

GemStone does not have any relational database capability built into the system. It does, however, provide the capability to extract and utilize data from relational database systems that have an SQL interface.

Objects in GemStone are grouped into collections, and the system provides several predefined types of collections. These collections can be ordered or

unordered, and various indexes can also be used to organize them. Objects are then grouped into segments for access control. Users can access all of the objects in a particular segment, or they cannot access any of them. The user can also specify clustering of the objects on the disk according to access patterns to minimize the disk access time.

Vbase. Vbase (Andrews 1987; Caruso 1987; Ontologic Inc. 1988; Thatte 1987) is a product of Ontologic Inc. The stated goals of Vbase are to provide an integrated language and database system that is general purpose and easily tailored to suit most any application. It originally aimed at the CAD and CAE markets, but currently claims to support any complex application with complex data.

Vbase currently runs under either the UNIX or VMS operating systems (with the claim that more are on the way). COP, the database language used in Vbase, is an extension of C. Interfaces to COP are also provided for C and C++.

Object SQL, an extension of the SQL relational interface, was recently added to Vbase. Unfortunately though, it is an add-on package and not fully integrated into the system. A user can either access the data through the COP interface or through the Object SQL

interface. There is no direct way provided to allow the user to access the data from a relational point of view and then switch to COP to work with the data retrieved through Object SQL. It is also not clear (through the information which we have been able to obtain) whether there are in effect two separate databases (an object database and a relational database which allows objects as abstract data types), or a single database which has two separate ways of entry.

VISION. VISION (Caruso 1987; Caruso 1988) is a product of Innovative Systems Techniques, Inc. The basic goal of the system is to provide a single, extensible environment for managing persistent data and the procedures that access and maintain it. This single environment is said to avoid the problems of separate database host languages. However (as far as we have been able to determine), there is no way to interface outside object-oriented languages to the database system.

The target application domains of VISION are investment research, market research, and medical information systems. This makes VISION different in that most of the other object-oriented database systems that we have studied are aimed more at the CAD/CAM and CASE markets.

To support these application domains, VISION is designed to support large statistical databases. Time is modeled by keeping a history of an object. Alternative scenarios are supported by allowing objects in the database to serve as prototypes for new (possible future) instances.

Two types of prototypes are supported, new and specialized. A new instance starts out as a copy of an existing one (the prototype). Changes to either the new or existing instance have no effect on the other. A specialized instance starts out as a pointer to the existing one. Changes to the existing instance will affect the specialized instance, but changes to the specialized instance have no effect on the existing one. A specialized instance can be thought of as always using the current object as its prototype, while a new instance can be thought of as using a specific version of the current object as its prototype.

Objects in VISION are stored in collections. Messages can be sent to individual objects or to collections of objects. A message to a collection operates on the elements of the collection in parallel.

One final note on VISION. It is claimed that instances belong to multiple classes. However, this does not mean that instances may have a form of multiple inheritance from other classes. What it means is that

one can move up the hierarchy and treat an instance of one class as though it were an instance of an ancestor of that class.

Other Object-Oriented Database Systems. There are a few other commercial and research systems that we have discovered with somewhat limited information available. In this section we present the information that we have been able to uncover.

Coral3 (Merrow 1987) is a research system being developed at the System Concepts Laboratory of the Xerox Palo Alto Research Center (Xerox PARC). It is essentially an extension of Smalltalk to allow for shared persistent objects among multiple users.

G-Base (Graphael 1988) is a product of Graphael, a company with its corporate headquarters in France. The limited information that we have on this system was obtained directly from Graphael personnel at the OOPSLA'88 conference that was held recently in San Diego, California. G-Base is aimed at CAD/CAM systems, command and decision support systems, production information systems, documentation management systems, simulation systems, training systems, and AI applications. There are a few production applications using G-Base in Europe, and Graphael is just now attempting to move into the U.S. market.

Iris (Fishman 1987) is a research prototype database management system being developed at Hewlett-Packard Laboratories. It is intended to meet the needs of CAD/CAM systems, CASE systems, and office information systems. Three interfaces are being developed for Iris. The first interface is called C-Iris, an extension of C. Another interface is OSQL (Object SQL), an object-oriented extension of the relational database language SQL. The third interface is called the Inspector. It is an extension of a Lisp structure browser, and is intended to eventually become a graphical interface. These three interfaces are separate and distinct ways of accessing the data, in much the same way that two interfaces have been provided for the Vbase system.

ORION (Kim 1987) is a prototype object-oriented database system being developed by the Microelectronics and Computer Technology Corporation. It is intended to support the data management needs of an expert system development environment.

POSTGRES (Stonebraker 1986), which stands for post INGRES, is being developed as a successor to INGRES (a relational database system implemented in 1975-1977 for research purposes). The goals of POSTGRES are to:

(1) provide better support for complex objects,

(2) provide user extendibility for data types, operators, and access methods,

(3) provide facilities for active databases (i.e., alerters and triggers) and inferencing including forward and backward chaining,

(4) simplify the DBMS code for crash recovery,

(5) produce a design that can take advantage of optical disks, workstations composed of multiple tightly-coupled microprocessors, and custom designed VLSI chips, and

(6) make as few changes as possible (preferably none) to the relational model.

Summary. All commercially available object-oriented database systems that we have studied organize objects into some type of a collection. The importance of this will be seen in the discussion of our system, the Relational Object-Oriented Management System (ROOMS), in Chapter 2. ROOMS is designed in such a way that it could be added to any existing object-oriented database system that allows user-specified collections of objects.

We believe that the following quote from Thatte (1987, 13) verily sums up the current status of object-oriented database systems:

> Relational database technology took over ten years before being accepted by the market. Object-oriented database technology will need five to ten years to transition from its current status of "proof of concepts" to the status of "full commercial systems."

CHAPTER 2

THE RELATIONAL OBJECT-ORIENTED MANAGEMENT SYSTEM (ROOMS)

As mentioned in Chapter 1, existing object-oriented database systems deal with collections of objects. In ROOMS, a relation is treated simply as a specific type (i.e., a subclass) of collection. As such, ROOMS could be added to any commercially available object-oriented database system that allows user-defined collections. ROOMS could also be implemented in any object-oriented language which has appropriate I/O commands (such as some form of random access), in which case ROOMS can be considered to be a complete database management system in its own right.

This chapter contains a discussion of the potential benefits of using ROOMS and an overview of the general design of ROOMS. This is followed by a detailed examination of the problems encountered in the development of ROOMS. It is then shown that these problems are not unique to ROOMS, that they apply to any large and complex object-oriented application. A more detailed description of the implementations of ROOMS (in PC Scheme and EOOPS) is given in Chapter 4.

Why ROOMS?

There are several benefits to a system such as ROOMS. Since the system makes no distinction between simple objects and more complex objects, both traditional and nontraditional applications can be supported with a single database management system.

Different methods can be used for different users. For instance, user-1 might see one display of an object while user-2 would see a completely different display of the object. Or, user-1 could compare objects in one way while user-2 compares them in another way. System security could also be built into the various methods - a user may see only part of an object or may not be able to view the object at all; authority to change the data could be restricted too.

Providing different methods for different users is a way of implementing views or subschemes of the data. A view is an abstract model of a part of the data (Ullman 1982). For example, the manager of a department may be able to see all of the data stored in the records of the employees in the department, including their salaries, while the employees in the department may have the information about salaries deleted from their view of the data.

An employee's age is also usually implemented as an abstraction - the date of birth of the employee is

normally stored in a record, not the age. Methods can then be provided to display either the age or the date of birth, as appropriate.

It is also a fairly simple matter to extend conventional applications developed in ROOMS to include more complex data. For example, a digitized photograph could be added to an otherwise conventional personnel record. Similarly, a real estate database could have the information on houses augmented by blueprints and maps.

Now, consider the real estate database example in more detail. Any conventional database system should be capable of providing the necessary facilities and constructs to define a simple real estate database. Basic information about the real estate, such as cost, lot size, school districts, square footage, number of bedrooms, and number of bathrooms, could easily be stored in any database system. A query system could then select entries based on any number of constraints on this information.

Non-conventional information, such as pictures or blueprints of the house, could not be added to the typical conventional database system. In ROOMS, however, any object can be added to the database. Therefore, digitized pictures of the house and drawings representing its blueprints could simply be added to the

record. Views could then be employed so that when looking at a list of records, only basic information is seen, and when looking at an individual record, the pictures and drawings of the house are also seen.

Adding a map showing how to get to the house is another problem, even with ROOMS if it has been implemented in an object-oriented language rather than added to an object-oriented database system. A simple map could be added to each record, but it would not be easy to show the "big picture." However, if ROOMS has been added to an object-oriented database system, then a detailed map of the area could be developed outside of ROOMS. The record in ROOMS could then provide some form of coordinates that would indicate where in this detailed map the particular house was located - the object-oriented database system could then be used to see the general location of the house, a detailed map of the immediate neighborhood, or anything in between.

In summary, we believe that the implementation of a relational database scheme within a more general object-oriented database system is a very important milestone for object-oriented programming. It helps to show that OOP is not just for applications that are not feasible with conventional languages. Even more traditional applications can benefit from OOP without sacrificing existing techniques for manipulating the data.

The General Design of ROOMS

Once again, the basics of the system are actually fairly simple. A relation is an object that is a collection of records. A record is an object that is a collection of fields. And a field is an object that is an instantiation of a user-defined class. The class definitions for these objects will now be examined in more detail.

The Class Database

The class database is defined with a single instance variable, named members. This instance variable contains a list of all of the relations that are in the database, e.g., '(R S T) would indicate that the database has three relations: R, S, and T.

Since database is a class, multiple instances (databases) may be created. Currently a global variable called ROOMS is used to indicate which database is in use, although there is no reason why multiple databases could not be in use simultaneously.

Various methods are defined for the class database. Display-yourself simply returns a list of the relations in the database. Add-member and remove-member add and remove relations from the database, respectively.

The Class Relation

The class relation is defined with two instance variables: members and record-type. Members contains a list of all of the records in the relation, e.g., '(e1, e3, e5) would indicate that the records e1, e3, and e5 are in the relation. Record-type contains the (user-defined) class-name of the type of records that can be members of that particular relation.

Several methods have been defined for the class relation. Display-yourself displays each record in the relation. Add-record and remove-record add and remove records to/from the relation, and get-records displays a list of all the records in the relation.

The five basic relational operations (union, difference, selection, projection, and Cartesian product) are defined as methods of relations. Two additional operations, intersection and fast-intersection, are also defined.

The union, difference, and fast-intersect methods were relatively easy to implement. They use the equal-to? methods (provided for the class Record and required for each user-defined class) to determine which records to select for the result relation. The intersect method was also relatively easy to implement, as it uses the "classic" formula of (R - (R - S)) for R intersect S.

The select method was more difficult to implement. At this time, records can only be selected by entire user-defined objects. That is, if OBJ1 is the name of an object that is part of a record, then selections can only be made based on equality with the value of OBJ1; selection is not allowed on equality with only a part of the value of OBJ1.

The project and Cartesian-product methods both presented special problems. Since individual record structures are defined as separate classes, the record structure for the result of project and Cartesian-product is different from that of the original relation(s). This may produce a new record structure that needs to be defined, or it may produce one equivalent to a previously defined structure. Currently, a new record structure is built unless the user specifies otherwise.

The Class Record

The class record is defined with no class variables or instance variables. It is intended to provide methods to user-defined records (using Smalltalk terminology, it would be implemented as an abstract class, providing a standard protocol for user-defined records). Several such methods are provided.

The display-yourself method concatenates the result of display-yourself for every variable and superclass of the record. The equal-to?, greater-than?, and less-than? methods make the appropriate comparisons by using the methods with the same names for every superclass of the record. The copy-from method makes copies of records. And the meet-select-criteria? method determines if a record meets the selection criteria set forth by the select method of the class relation.

User-Defined Classes

Since ROOMS is a relational system, the objects must be stored in records. User-defined record classes must either have the class record as an ancestor or have all of the methods that would have been provided by the class record to work properly with the class relation.

The only limits on user-defined classes in ROOMS are the user's imagination and, of course, any limitations imposed by the host language and the hardware that is being used. Once a user-defined class is specified, objects of that type can be instantiated and stored in the database.

Various methods are required for the user-defined classes to work properly with the relation and record classes. Display-yourself is used by the display-

yourself methods of the relation and record classes. Similarly, a copy-from method is also required.

Equal-to?, greater-than?, and less-than? methods are also required, although these may simply return false if it is not appropriate to say that one object is equal to, greater than, or less than another object. Note that this has the potential of changing our "normal" logic in that it is possible for two objects to be not equal to, not greater than, and not less than; and that less than or equal to and not greater than may both be false!

All of these required methods are provided by ROOMS (in the generic class in the PC Scheme version and in the default-methods class in the EOOPS version). These generic or default methods simply operate upon all of the variables and superclasses that make up a user-defined class (see the section on Genericity in this Chapter and also Chapter 4 for implementation details and a comparison to other approaches). They may not meet the needs of all user-defined classes, but they do provide a starting point.

The notion of methods which are required by user-defined classes is similar to the idea of subclass responsibility as used in Smalltalk applications (as discussed in Chapter 1). However, there is no requirement that the user-defined classes be a subclass

of any class supplied by ROOMS, only that these required methods be provided so that instances of the user-defined class may be properly manipulated by ROOMS.

Problems Encountered in Developing ROOMS

Several problems were encountered while developing ROOMS in the object-oriented language PC Scheme. It will be shown later in this chapter that these problems are not unique to either ROOMS or PC Scheme. In fact, most of the problems are presented in a form that is not dependent upon either ROOMS or PC Scheme.

"Conventional" Inheritance Problems

Most of the problems encountered were due to the way in which inheritance has been implemented in "conventional" object-oriented languages. It is easy to think of a subclass as "everything that its superclass is, plus some newly defined variables and methods." However, a subclass is actually "everything that its superclass is, plus some newly defined variables and methods, except that some of the inherited variables and methods have been redefined." If the class designer is fully cognizant of all the inherited variables and methods (from all of the ancestors), then there is no problem. However, by simply reusing an inherited variable or method name by "accident," it is quite

possible to redefine a variable or method without realizing it - this is the root of most of the problems.

Variable Name Conflicts. Name conflict problems are illustrated in Figure 1. Let us assume that Object-B is an instance of Class B. Object-B will have the variables: Q, R, S, and T. Hopefully, the designer (and users) of the system will be aware that Object-B's Q is the Q defined in Class B rather than the Q defined in Class A.

It could be rightfully argued that this is not a problem, rather it is what should be expected - the variable Q has simply been redefined for instances of Class B. However, this means that the designer of Class B must know all that there is to know about the variables inherited from Class A. We cannot simply define Class B to be everything that Class A is plus something new without carefully examining the definition of Class A.

Figure 2 shows how this situation gets even more involved as we move further down the class hierarchy. If Object-C is an instance of Class C, it will have five variables: Q, R, S, T, and U. The S, T, and U are as defined for Class C, the Q as defined for Class B, and

Class A

Superclasses: none

Variables: Q, R, S

Methods: X, Y

Class B

Superclasses: A

Variables: Q, T

Methods: X, Z

Figure 1. Single Inheritance in a Two-level Hierarchy.

Class A

Superclasses: none

Variables: Q, R, S

Methods: X, Y

Class B

Superclasses: A

Variables: Q, T

Methods: X, Z

Class C

Superclasses: B

Variables: S, T, U

Methods: W, Z

Figure 2. Single Inheritance in a Three-level Hierarchy.

the R as defined for Class A. Once again, the designer of the system must be aware of all of these definitions.

Method Name Conflicts. In addition to variable name conflicts, we also have method name conflicts. Again consider Figure 1. Object-B will have three methods: X, Y, and Z. The X and Z will be as defined for Class B, and the Y is as defined for Class A. If Y uses the variables Q, R, and S, it will get R and S as defined for Class A and Q as defined for Class B. If Y uses method X (by sending self the message X), it will now use the X that has been defined for Class B. It is up to the designer of Class B to make sure that the variable Q and method X defined for Class B will work properly with the method Y inherited from Class A.

This situation also gets more involved as we move down the class hierarchy. Referring again to Figure 2, Object-C will have four methods: W, X, Y, and Z - W and Z as defined locally for Class C, X as defined for Class B, and Y as defined for Class A. And now if Y uses variables Q, R, and S, it will get Q as defined for Class A, and S as defined for Class C.

An additional problem with some object-oriented languages is an inability to build upon inherited methods when a locally defined method has the same name. Referring again to our example from Figure 1, Object-B

has three methods: X, Y, and Z. The X is as defined for Class B, but there is no way to access the method X defined for Class A, which should have been inherited by Class B. Even if the new method X (defined for Class B) is intended to be an extension of Class A's method X, we cannot simply say to perform Class A's X and then do something more.

However, a few object-oriented languages, such as Flavors (Moon 1986) and Trellis (Schaffert 1986), do allow inherited methods to be called by using the class name and the method name. As will be seen in Chapter 3, this is the approach that we have taken with the Encapsulated Object-Oriented Programming System (EOOPS). Smalltalk (Borning 1982) and CLOS (Bobrow 1988) also allow the use of super.method-name, but this is not sufficient for multiple inheritance, as methods with the same name may be inherited from different superclasses.

Multiple Inheritance. Multiple Inheritance adds to the confusion that has been created. Consider Figure 3. If Object-F is an instance of Class F, it has five variables: Q, R, S, T, and U. The Q and U are obviously as defined for Class F, the R as defined for Class D, and the T as defined for Class E. But what about the S? There is an S defined for Class D and another S defined for Class E. Which definition of S is inherited?

Class D		Class E	
Superclasses:	none	Superclasses:	none
Variables:	Q, R, S	Variables:	Q, S, T
Methods:	X, Y	Methods:	X, Z

Class F	
Superclasses:	D, E
Variables:	Q, U
Methods:	Y

Figure 3. Multiple Inheritance in a Two-level Hierarchy.

Most existing object-oriented languages that support multiple inheritance resolve situations such as this by some predetermined formula, usually based on the order in which the superclasses are presented. Assuming that we would start with the superclass that is listed first, the S will be as defined for Class D.

Similarly, Object-F will have three methods: X, Y, and Z. The Y is as defined for Class F and the Z is as defined for Class E. Using the same conflict resolution rules for methods that we used for variables, the X will be as defined for Class D. Now assume that method Z (originally defined for Class E) uses the variables Q, S, and T, and the method X. For Object-F, the variable Q will be as defined for Class F, S as defined for Class D, and T as defined for Class E, and the method X will be as defined for Class D.

It can still be argued that we are simply redefining both variables and methods, and that it is up to the designer to ensure that the changes are compatible with all inherited variables and methods. Even with all of these problems, has any real harm been done?

Compounding the Problem - Modifying Class Definitions. All of these problems get even worse when we consider the effects of changing class definitions. Adding to

the definition of a class presents no real problems for any of its descendants other than for persistent objects. Removing or redefining variables or methods, however, can have a "rippling" effect, requiring that all of the descendants be checked for compatibility with the changes of the ancestor.

Once again consider Figure 1. If the definition of the variable Q or the method X of Class A is changed or even deleted, there is no effect upon the definition of Class B. If a variable or method is added to the definition of Class A, then that variable or method has in effect also been added to the definition of Class B. However, if we change the definition of either of the variables R or S, or the method Y, then the methods X and Z of Class B must be carefully checked to ensure that they will still work properly with the modified variables and methods of Class A.

This process of deciding which descendants must be modified is not as easy as it might first appear. Although we talk about superclasses and subclasses, most object-oriented languages only define superclasses; there is no way to directly dete ne what subclasses a class has by simply looking at a program listing (as only the superclasses are listed). Therefore, we must generate the class hierarchy to determine what the descendants of a class are. Note, however, that it is

possible to build this feature into an object-oriented language; the GemStone database system (Penney 1987) does check the subclasses to determine if a proposed change to the variables of a class will cause problems with that class' descendants.

Database Record Structure

There were several design issues in developing ROOMS, mainly as to when to use subobjects and when to use dependent objects. We generally prefer subobjects as they are individually addressable while dependent objects are not. This allows the user more flexibility in accessing objects - either the composite object can be told to send a message to one of its subobjects, or the message can be sent to the subobject directly. This probably "opens the door" to potential user abuse, but it also allows for the potential sharing of subobjects.

In particular, the record structure in ROOMS was a question mark. We originally treated the fields of the record as subobjects due to the potential name conflicts of multiple inheritance. If multiple inheritance without these problems were available, however, what form should a record take? It could be a composite object, consisting of either subobjects or dependent objects, or all of the fields could be superclasses under multiple inheritance.

As previously mentioned, it has been stated that multiple inheritance should only be used when the new class is the sum of the behaviors of the superclasses (Halbert 1987). Is a record the sum of the parts (fields)? Or is it the sum of the behaviors of the fields? It could probably be argued either way, but we would prefer to use multiple inheritance. For example, the message

(record change-zip 32951)

seems to be easier to understand and more straightforward than the message

((addr of record) change-zip 32951)

especially when one considers that the level of nesting of subobjects/dependent objects could be very deep, for example

((a of (b of (c of (d of ...)))) message argument)

We believe that a user of ROOMS should only be concerned with accessing relations and records. A user should not have to know about how the record is constructed in order to gain access to the data that makes up the record. Therefore, messages should be sent to the record rather than to its parts, and as such we should use multiple inheritance.

Relational Operations

The union and difference operations were relatively simple and straight forward to implement, and as such they were the first operations implemented. The result of these operations is a relation which is of the same type (class) as the original relations, so the only concern was with the operation itself.

The select operation was somewhat more difficult. The result relation is still of the same type as the original relation - the problem is in determining what to allow as selection criteria. At this time, selection is allowed only on whole objects (i.e., subobjects).

The project and Cartesian product operations presented special problems. Project uses only a part of each record while Cartesian product combines two record structures. If the effects of these operations are to be more than temporary (e.g., the effect of a "display" is temporary and the effect of an "assignment" is permanent), then we are faced with building new record structures. Either the user will have to define these new structures in advance, or the system will have to define them as needed. It is preferable to have the system define the new structures dynamically. This means that what we really need is a dynamic class builder, a proper metaclass to which a message can be

sent to build the appropriate new record structure class.

Genericity

Generic code is a potential answer to part of the name conflict problems of inheritance, but unfortunately PC Scheme does not support genericity. However, since PC Scheme is LISP based (which allows the output of a routine to be another routine), some "simple" generic routines can be developed. And by applying some structure to choosing variable names, "semi-generic" methods can be used.

The use of "semi-generic" code is illustrated in Figure 4. In this example, the only method that Class New-Object has is Display-Yourself, which will use the class variable New-Object-Display-List. The method, which is inherited from the Class Object, will be using a class variable that the Class Object does not even have. Note that in the display list: (1) the instance variables can appear in any order, (2) instance variables may appear more than once, and (3) instance variables may even be left out of the list.

This generic approach, especially the idea of "semi-generic" code, is comparable to the idea of subclass responsibility in common protocols in Smalltalk (Johnson 1988; Pinson 1988) and virtual methods in C++

```
                Class Object
Superclasses:              none
Class Variables:
     Object-Display-List    value: null
Instance Variables: none
Methods:
     Display-Yourself
       For x in
        (concatenate (class-of self)
                     '-Display-List)
         display x

                Class New-Object
Superclasses:         Object
Class Variables:
     New-Object-Display-List       value: '(Q, R, S)
Instance Variables: Q, R, S
Methods:              none
```

Figure 4. "Semi-generic" Code.

(Stroustrup 1986b; Wiener 1988). The general idea of all three approaches is essentially the same - the superclass can provide methods that can be used by the subclass, but these methods may need specific information from the subclass (or the instance of the subclass) in order to function properly. The most commonly used approach in Smalltalk and C++ is to require that the subclass include specific methods to provide this information, whereas in ROOMS we have required that the subclass include specific class variables to provide this information.

There are three main reasons that we chose to require specific class variables in the subclass rather than specific methods. First is that it seems (to us) to be easier to simply include "extra" class variables in a class definition rather than to encode "extra" methods so that the superclass methods will function properly. Secondly, class variables may be changed at any time to alter their values (and therefore also alter the result of the inherited method), while a subclass method would have to be reimplemented to change the value that it returns. The third reason is directly related to our use of PC Scheme as our main test language - in PC Scheme direct access of a class variable is much more efficient (and therefore faster) than executing a method to obtain the same information.

Why These Problems Are not Unique to ROOMS

Most of the problems of name conflicts with inheritance in "conventional" object-oriented languages were presented in general terms. The design issues of dependent objects versus subobjects versus multiple inheritance are applicable to building any system of class definitions. The project and Cartesian product operations are specific to relational systems, but the more general ideas of dynamic class building and metaclasses are not. The idea of generic methods that can be instantiated for various classes is also not unique to ROOMS - any application could benefit from this.

None of the examples used in describing the problems discussed in the previous sections are specific to PC Scheme. Other OOP languages may use a different set of rules for determining which variable or method to use in the case of duplicate names, and some do allow an inherited method to be accessed even if a new method with the same name has been defined. Some OOP languages do not even allow multiple inheritance. The design decision of whether to use dependent objects or subobjects is also independent of the specific language used. And PC Scheme is not the only OOP language that lacks a dynamic class builder.

In summary, none of the languages that we have studied provide easy solutions to all of the problems that we encountered. This is the main reason that we designed the Encapsulated Object-Oriented Programming System (EOOPS), which is discussed in the next chapter.

CHAPTER 3

THE ENCAPSULATED OBJECT-ORIENTED PROGRAMMING SYSTEM (EOOPS)

The underlying philosophy of EOOPS is that encapsulation should be fully preserved, even with inheritance - a subclass should not know about or be dependent upon the specific definition of class variables, instance variables, or methods of any of its ancestors. A subclass is restricted to the external interface of its superclass(es), with no direct access of any inherited variables.

It should be noted that we cannot claim to be the originator of all of these ideas. Our main motivation for EOOPS came from the article "Encapsulation and Inheritance in Object-Oriented Programming Languages" (Snyder 1986), and from our frustration with the way that inheritance is implemented in PC Scheme (Texas Instruments Inc. 1987a,b), the main test language used in developing ROOMS.

It should be remembered that EOOPS is not intended to be a complete language. As mentioned in the introduction to this dissertation, only those features unique to an OOP language were included. Thus, EOOPS

might be considered a "bolted-on" OOP package that could be added to any conventional language, along the lines of SCOOPS, the object-oriented part of PC Scheme, or it could serve as the core of a language in which OOP principles are "built-in."

This chapter discusses the potential benefits of EOOPS, and then presents EOOPS from a design point of view - what was done and why. The EOOPS Language Reference Manual is provided in Appendix A.

An Encapsulated Form of Inheritance

In EOOPS, inherited instance and class variables are only accessible through inherited methods. If variables with the same name are inherited from different ancestors, then distinct copies of those variables are maintained, each accessible only through the appropriate methods that are inherited with it.

The Internal and External Interfaces

Two interfaces are defined for each class - an internal interface and an external interface. As commonly used in other OOP languages, the external interface is the set of methods available for use by (users of) the instances of a class. In EOOPS, it consists of the methods defined locally for the class and the inherited methods which can be referred to

unambiguously by using a "first-come first-served" resolution scheme.

The internal interface is the set of methods which are available for use by those methods which are defined (i.e., not inherited) for a class. It consists of the methods defined locally for the class and all of the methods in the external interface of each superclass (not to be confused with each ancestor of the class). Method name conflicts in the internal interface are avoided by concatenating the name of the superclass with the name of the method (i.e., superclass-name.method-name).

In the "first-come first-served" resolution scheme which is used in defining the external interface of EOOPS, the method that will be chosen in the case where several methods with the same name are inherited will be that of the "first" parent (based on the textual order in which the superclasses are listed) which defines that operation (Snyder 1986). This resolution scheme is the one that is used in PC Scheme.

For single inheritance, this resolution scheme effectively assumes that all inherited methods will become part of the new class' external interface, unless the new class defines (redefines) a method with the same name as an inherited method. In this case, the inherited method is no longer a part of the external

interface - it has been replaced by the locally defined method of the same name.

For multiple inheritance, this assumption holds true only when methods with the same name are not inherited from different superclasses. There are several possibilities for determining which inherited method(s) will be in the external interface in this case, including choosing the "first" one that is found, choosing one other than the "first," choosing all of the inherited methods (either in some particular order or in parallel), or choosing none of the inherited methods (i.e., the inherited methods with the same name would be available in the internal interface only).

In EOOPS, the "first" method that is found will be chosen for the external interface. If this is not what is desired, then the method may be redefined by the inheriting class. Additionally, any of the inherited methods may be specifically excluded from the external interface by using the external-delete-method command.

It should be noted that any method, whether inherited or defined locally, can be specifically excluded from the external interface by using the external-delete-method command. Therefore, the designer of the class has the final decision as to which methods will be in the external interface (for more information

on this command, see the section on External-Delete-Method Command in this chapter or refer to Appendix A).

When using inherited methods, the class definition is effectively tied to its superclass(es). That is, if the class is redefined to change its position in the hierarchy (by deleting or changing superclasses), then the methods which were inherited will no longer be available (unless a new superclass provides appropriate methods with the same name).

However, this is a problem in any object-oriented language, not just in EOOPS. It is up to the designer (or redesigner) of a class to ensure that its external interface is maintained for maximum compatibility with its old definition. Inherited methods which were available in the external interface must now be provided, either from the modified class itself or from a new superclass, to accomplish this.

While this would maintain the external interface, the internal interface is still a problem. If a method defined by a class calls an inherited method using the superclass-name.method-name format, then an error will occur if superclass-name is deleted as a superclass. However, all of the methods in the external interface are also in the internal interface, so message calls using the superclass-name format are only necessary if that method has been redefined or if more than one

method with that name has been inherited. Thus, this problem can be minimized by using the superclass-name format only when necessary.

Having an external and an internal interface for a class is similar to the notions of public and private provided by C++ (Stroustrup 1986b). If inheritance is not considered, then the external interface of EOOPS is equivalent to the set of methods which have been declared to be public in C++, and the internal interface of EOOPS is equivalent to the union of the public and private sets of methods in C++.

With inheritance, however, the two approaches begin to differ. In EOOPS, all of the locally defined methods and all of the inherited methods are in the internal interface. All of the locally defined methods are also in the external interface, but the only inherited methods which are included are those which can be referred to unambiguously. In C++, a class is inherited (actually, in C++ terminology, we say that a class is derived from another class rather than that it inherits from another class) as either public or private, in which case only the public methods of the superclass become either public or private in the subclass (the private methods of the superclass are not directly accessible to the methods of the subclass in C++).

C++ also includes friend functions (methods) and protected sections of the class definition (Micallef 1988; Stroustrup 1986b; Wiener 1988). These are always directly accessible to the subclass. Similarly, Trellis allows subtype-visible operations (Micallef 1988; Schaffert 1986). These schemes provide a way to make methods available to the subclasses but not to the instances of a class. EOOPS also provides for subtype-visible definitions with the generic-class concept - this is discussed further in the Genericity section of this chapter.

Single Inheritance in EOOPS

Single inheritance in EOOPS can be illustrated by once again using Figure 1. Now Class B has the variables Q and T defined locally and a set of variables {Q, R, S} which it inherits from Class A. The internal interface of Class B consists of the methods X and Z defined locally, and the methods A.X and A.Y which are inherited from Class A (i.e., the external interface of Class A has become a part of the internal interface of Class B). The external interface of Class B consists of the methods X and Z defined locally, and Y which resolves to A.Y. The method A.X is not a part of the external interface (due to a name conflict with the X

defined locally) and is therefore not available to the descendants or instances of Class B.

The methods X and Z defined for Class B can only access the variables Q and T which are also defined locally. To access any inherited variables requires that an inherited method be used. And when an inherited method accesses a variable by name, it will access the appropriate variable inherited along with it, e.g., if method Y inherited from Class A accesses variable Q, it will use the variable in the set {Q, R, S} that was also inherited from Class A.

It should be noted, though, that (the designer of) a class need only be concerned with the behavior of inherited classes (i.e., the methods that are inherited) and not with the variables that are inherited. An inheriting class only knows that a set of variables (possibly empty) has been inherited. It does not know how the inherited methods are implemented or what variables they access - it only knows in general terms what the inherited methods do (i.e., their behavior).

If an inherited method calls upon other methods, it can only use methods that it knew about when it was defined, e.g., if method A.Y (inherited from Class A) calls method X, it will use the method X which was also inherited from Class A; when we call an inherited method, we change to its internal interface until it is

finished executing. If this is not what is desired, then a new method should be provided, i.e., define a new method Y for Class B which uses the method X defined for Class B. However, Class B should only know what methods are inherited and what these methods accomplish - it should not know (or even be concerned with) how the methods are implemented and what methods or variables they use.

Once again refer to Figure 2 to see how this encapsulated form of inheritance works as the inheritance hierarchy is expanded from two levels to three levels. Object-C now has the variables S, T, and U defined locally, and a set of variables {Q, T, {Q, R, S}} inherited from Class B. The internal interface of Class C consists of the methods W and Z defined locally, and B.X, B.Y, and B.Z (the external interface of Class B). The external interface of Class C consists of the methods W and Z defined locally along with the methods X and Y which resolve to B.X and B.Y (note that Y was inherited by Class B from Class A).

As in the two-level hierarchy example, inherited variables are only accessible by using the methods which were inherited along with them. In the three-level hierarchy, however, the inherited set of variables {Q, T, {Q, R, S}} contains an inherited set of variables {Q, R, S}. The variables in this "internal" set, originally

inherited by Class B from Class A, are only accessible by using those methods inherited from Class B which were either inherited from Class A or use a method that was inherited from Class A.

Once again though, as in the two-level hierarchy, a class need not be concerned with the inherited variables or which inherited methods access these variables. We do not even need to know that it is a three-level hierarchy. The actual implementation of the class which we are inheriting from is unimportant to us. We need only be concerned with its behavior (i.e., its external interface).

The methods A.X and A.Y are not available for direct use by Class C (i.e., they are not in either the internal interface or the external interface of the class) as they were not included in the external interface of Class B. If we allow either an instance of Class C or a method of Class C to directly use the method A.X or A.Y inherited from Class A, then we have again violated encapsulation. All that Class C should know is that it has variables S, T, and U, methods W and Z, and a superclass B. It should also know that it has the methods B.X, B.Y, and B.Z and a set of variables that it inherited from its known superclass B. If Class C knows that it has methods A.X, and A.Y, then in some

way it knows "too much" about its ancestry - it knows more than what it was told in its definition.

Multiple Inheritance in EOOPS

Multiple inheritance in EOOPS is illustrated using Figure 3. Now Class F has the variables Q and U which are defined locally, a set of variables {Q, R, S} inherited from Class D, and another set of variables {Q, S, T} inherited from Class E. The method Y defined in Class F can only directly access the variables Q and U defined in Class F - to access any inherited variable requires that an inherited method be used. Also, if an inherited method accesses a variable by name, it will access the appropriate variable inherited from the same class, e.g., if method Z inherited from Class D accesses variable Q, it will use the variable in the set {Q, R, S} that was also inherited from Class D.

If an inherited method calls upon another method, then it will only use the method that is inherited along with it, e.g., if method Z inherited from Class D calls method X, it will use the method X inherited from Class D, not the method X inherited from Class E or some method X which could be defined for Class F. If this is not what is desired, then a new method should be provided, i.e., define a new method Z for Class F which uses some other method X. However, the designer (and

users) of Class F will only know what methods are available for use; they need not be concerned with how these methods are implemented.

If methods with the same name are inherited from different superclasses, then each is still available. This is accomplished by concatenating the name of the superclass with the name of the inherited method, e.g., to specifically reference class D's method X, use D.X.

Multiple Inheritance From a Single Superclass

In Figure 5, Class G inherits from Class A twice-once directly and once indirectly (from Class B). Class G should only know about the direct inheritance of Class A. The set of variables and methods from Class A that is inherited indirectly through Class B should only be available through the methods inherited from Class B. If Class G knows that it has inherited more than once from Class A, then it knows more than what it has been told in its definition.

Thus, if Object-G is an instance of Class G, it will have the variables S, T, and U defined locally, a set of variables {Q, R, S} inherited from Class A, and a set of variables {Q, T, {Q, R, S}} inherited from Class B. The internal interface of Class G will contain the methods W, Z, A.X, A.Y, B.X, B.Y, and B.Z. The

Class A

Superclasses: none

Variables: Q, R, S

Methods: X, Y

Class B

Superclasses: A

Variables: Q, T

Methods: X, Z

Class G

Superclasses: A, B

Variables: S, T, U

Methods: W, Z

Figure 5. Multiple Inheritance with an Ancestor Appearing Twice in the Hierarchy.

external interface will contain the methods W, Z, and Y (which resolves to A.Y).

Figure 6 shows how this idea is expanded to allow multiple inheritance directly from a single superclass. Class I will inherit three sets of variables {Q, R, S} and three sets of methods X and Y. The methods are referred to as H.1.X, H.2.X, etc. The individual sets of variables can only be referred to by the appropriate methods that are inherited along with them. The usage of a structure such as this may be limited (one example is a target, made up of several circles of alternating colors), but when encapsulation is preserved, it is possible.

The Metaclass

A metaclass has also been included in EOOPS. The metaclass has a dynamic class building method available which is similar to the typical "define-class" method (operation) that exists in most OOP languages. A message can now be sent to the metaclass giving the name of a new class and the names of superclasses, class variables, and instance variables for the new class. The metaclass then produces a new class with the appropriate structures. Users can also add methods to the metaclass in order to meet the various needs of specific applications.

Class H

Superclasses: none

Variables: Q, R, S

Methods: X, Y

Class I

Superclasses: H, H, H

Variables: none

Methods: X

Figure 6. Multiple Inheritance from a Single Ancestor.

Genericity

Generic code is also a feature of EOOPS. This is really nothing new in itself, as various forms of genericity are available in many non-OOP languages. However, most discussions of genericity and OOPS are either comparisons (Meyer 1986) or they are concerned with showing how inheritance can be simulated using genericity (Touati 1987). They have not addressed how genericity itself can be applied to enhance the existing features of OOPS.

There are two types of generic code in EOOPS - generic methods and generic classes. A generic method is simply a method which can be instantiated by user-defined classes. A list of generic parameters is used to match up variable names within the generic method with methods and variables available to the user-defined class.

Generic class definitions are used to provide class variables, instance variables, and methods by name to a regular (i.e., not generic) class. When instantiated (as part of a user-defined class), a generic class is treated essentially the same as an inherited class in a conventional object-oriented language. Generic methods can be instantiated for generic classes. A generic class may not have any instances.

A generic class can be compared with the idea of the abstract class (Johnson 1988). The difference is that EOOPS provides generic classes as a construct rather than as a concept - an abstract class has no instances, but this is not enforced by the system, whereas EOOPS does not allow generic classes to have instances. A "regular" class in EOOPS could also be an abstract class if it had no instances.

Another function of generic classes is more closely related to the idea of having public, private, and subtype-visible partitions in some other object-oriented languages (Micallef 1988). The external and internal interfaces of a "regular" class in EOOPS provide the public and private sets of methods (all variables are private). The subtype-visible part has been placed into a separate definition altogether, that of the generic class.

Special Methods

Several special methods have been incorporated in EOOPS. These methods give the user information about classes and objects. Most of these special methods are similar to functions which are provided in PC Scheme.

Three such methods are get-superclasses, get-class-variables, and get-instance-variables. Sending one of these messages to a class will return a list of

superclasses, class variables, or instance variables, as appropriate.

Another special method is class-of-object?. This method simply returns the class of the object to which the message is sent. It returns nil if the "object" to which it is sent is not an object (i.e., it is a variable or constant).

The External-Delete-Method Command

As previously mentioned, the external-delete-method command is used to delete methods from the external interface only (i.e., the methods remain in the internal interface). There are two main uses for this command. First, as previously discussed, inherited methods may be specifically deleted from the external interface. Secondly, new methods may be defined for a class (which would normally be a part of both the internal interface and the external interface) and then deleted from the external interface.

This command can be used to give the equivalent of the public and private methods as defined in C++ (Stroustrup 1986b). The main difference is that a class definition in C++ declares each of the methods of the class as either public or private, while in EOOPS methods are not declared at all in the class definition. This means that new methods may be added to an EOOPS

class at any time without modifying the class definition.

Two different define-method commands could have been provided in EOOPS (one for methods to be in the internal interface and one for methods to be in the external interface), but this would not have allowed for the deletion of inherited methods from the external interface. Similarly, the class definition could have been extended to include a declaration of external and internal only methods, but this still would not have allowed for changing the status of inherited methods.

Regardless of the benefits of using a procedural approach (such as that used in EOOPS) versus the benefits of using a declarative approach, the end results are the same. The main advantage of having an explicit external-delete-method command is that the status of inherited methods in the external interface may be handled individually, rather than having to treat all of the methods inherited from a superclass as either public (i.e., as part of the external interface) or private (i.e., as part of the internal interface).

The Send Command

Send is used in the SCOOPS portion of PC Scheme to indicate that a message is being sent to an object. For example, (send obj1 message1) indicates that message1 is

being sent to obj1. However, most object-oriented languages use a format similar to (obj1 message1) to send message1 to obj1.

Although it is not specifically stated in the PC Scheme manuals (Texas Instruments Inc. 1987a,b), we are fairly sure that the keyword send is simply used as a "trap" for SCOOPS commands - it indicates to the compiler that a SCOOPS command is coming.

We have opted to include the keyword send in EOOPS for two reasons. The first is for purely aesthetic reasons - it is easier to pick out the messages in a program listing when the send keyword is present. This should be fairly obvious when scanning through Appendix C.

The second reason, however, is more substantial. Using the keyword send allows users to build their own trap. A macro can be written using another keyword (such as dbsend) which may perform some other function (such as error checking) before sending the message on to the object.

Special Note on Class Variables

Inherited class variables can be implemented in two different ways. Consider Class J in Figure 7. It has one class variable, Q, which is shared in both name and

Class J

Superclasses:	none
Class Variables:	Q
Instance Variables:	none
Methods:	none

Class K

Superclasses:	J
Class Variables:	none
Instance Variables:	none
Methods:	none

Figure 7. Inheriting Class Variables.

value by all instances of Class J. Class K has Class J as a superclass, so it inherits the class variable Q.

But is this variable Q now shared in both name and value by all instances of Class J and Class K? That would mean that if an instance of Class J changed the value of Q, it would also be changed for instances of Class K. It would also mean that if an instance of Class K (or any other descendant of Class J) changed the value of Q, it would also be changed for instances of Class J.

The other alternative is that Class K inherits a copy of the variable Q defined for Class J. Now, changing the value of Q for either Class J or K has no affect upon the value of Q for the other class. This is the approach that was chosen for EOOPS. Allowing an instance of a descendant class to change the value of an ancestor's class variable just does not seem to go along with the "spirit" of the strict encapsulation provided in EOOPS. Another possibility is to allow both kinds of class variables - this alternative is discussed in Chapter 5.

The Benefits of EOOPS

There are several benefits to using a system such as EOOPS. Encapsulation is strictly enforced, so that both the instances of a class and the subclasses of a

class are presented with the same interface. This greatly reduces the effects of modifying class definitions. The internal structure of a class (i.e., the variables) may be modified, without affecting clients (the users, either instances or subclasses) of the class as long as the external interface still appears the same (i.e., maintains the same functionality and message calling format).

For example, a class circle could be defined using a Cartesian coordinate system. With the strict form of encapsulation that is used in EOOPS, if the class circle is modified to use a polar coordinate system instead, the changes will have virtually no effect on any client of the class circle. No effect, that is, as long as appropriate modifications are also made to (the methods in) circle's external interface so that it appears to be the same as before.

One problem that still exists, even with EOOPS, is the effects that changing a class definition has on persistent objects (i.e., objects which last beyond a single session). Continuing with the class circle example, instances of the class circle and the instances of all of its subclasses which were created before circle's definition was changed will need to be modified if they are to be used with the new definition of circle.

Although EOOPS does not solve this problem, it does help, in so far as the subclasses are concerned. Since none of the variables of the class circle can be redefined by any subclass, the changes necessary for instances of subclasses of the class circle are the same as for instances of class circle. Furthermore, if the original external interface is maintained (or only added to), then the use of these methods within the subclasses will still be correct.

The "accidental" redefinition of inherited variables (i.e., redefining an inherited variable by reusing its name without realizing it) is no longer possible, so this problem is eliminated. Therefore, programmers need only be concerned with the external interface of inherited classes rather than their actual implementation.

The larger the programming project and the more programmers involved, the greater these benefits become. Consider a "fairly large" system being developed by 10 programmers. With EOOPS, each of the programmers can independently develop their part of the system, providing the others with only their particular external interface. Changes can then be made to the class definitions in any section of the system, with minimal effects on the other sections as long as the external interface is maintained. Only persistent

objects of classes which have a modified class as an ancestor will be affected.

Without an encapsulated form of inheritance, the effect of changing a class definition will "ripple" through the class hierarchy. Any descendant of the modified class that accesses inherited variables by name must also be modified. This means that no amount of testing of the modified class alone will guarantee that the system will still work properly. Now take our previously mentioned "fairly large" system with 10 programmers and turn it into a "very large" system with 100 (or more) programmers, and this problem gets even worse.

Unfortunately, this problem is not eliminated completely by EOOPS. Using generic classes to model generic concepts gives a class which does not have any instances, but the variables in the generic class are accessible by name in classes which instantiate the generic class (i.e., the definition of a generic class is subtype visible). Therefore, changes to a generic class may directly affect classes which instantiate it.

However, generic classes do allow generic concepts to be defined as such rather than implemented as an abstract class that appears the same as a regular (or concrete) class. It also means that all definitions which are subtype visible can be readily identified,

because only the definitions of generic classes are subtype visible.

Generic methods allow for functions which may be needed by several different (possibly unrelated) classes to share the code which defines the methods. This could reduce the amount of code in large systems, which would ease the task of maintaining the system.

Since the metaclass in EOOPS is treated as a "regular" class, methods may be added to it. This means that EOOPS could be used to test and implement new ideas for object-oriented systems by simply adding new methods to the metaclass.

We have now discussed (in Chapter 2) the basic design of ROOMS and (in this chapter) the basic design of EOOPS. In the next chapter we will compare and contrast the implementations of ROOMS in PC Scheme and EOOPS.

CHAPTER 4

THE IMPLEMENTATION OF ROOMS

As mentioned in Chapter 2, the basics of ROOMS are actually fairly simple. A relation is an object that is a collection of records. A record is an object that is a collection of fields. And a field is an object that is an instantiation of a user-defined class.

While Chapter 2 discussed ROOMS from a general design viewpoint, this chapter presents a comparison of the actual implementation of ROOMS in PC Scheme and EOOPS. To help simplify this discussion, the PC Scheme version of ROOMS will be referred to as PCS/ROOMS and the EOOPS version will be referred to as EOOPS/ROOMS. The actual code for these implementations is given in Appendices B and C.

Required Methods

Several methods are required for virtually all of the classes defined for ROOMS. "Generic" versions of these methods are provided in both PCS/ROOMS and EOOPS/ROOMS. User-defined classes may either use these generic methods, define their own versions of these

methods, or use some of the generic methods and define some of their own.

In PCS/ROOMS, these methods are defined in a class called generic. These methods are available if this generic class appears anywhere in the ancestry of a user-defined class (note that this is a "regular" class that is named generic, not a generic class as can be defined in EOOPS).

In EOOPS/ROOMS, these methods are defined as generic methods, which can then be instantiated as needed by the various classes. A generic class could have been used, but a "regular" class could not have been used because inherited methods would not be able to access locally defined variables.

The method display-yourself is required for all classes. It must provide some form of display of the object. The generic version simply displays the value of each class variable and instance variable of the object, along with all of the inherited class variables and instance variables.

Equal-to?, less-than?, and greater-than? methods are also required. They must be capable of comparing two objects, and return true or false. They may, however, simply return false if it is not appropriate to say that one object is equal to, greater than, or less than another object. Note that this has the potential

of changing our "normal" logic in that it is possible for two objects to be not equal to, not greater than, and not less than one another. The generic versions of these methods simply compares the values of each instance variable of the objects and then compares the values of each of the inherited instance variables.

A copy-from method is also required for each class. This method allows an object to make itself a copy of another object. The generic version sets the value of each instance variable of the first object to that of the second object, and then sets all of the inherited instance variables of the first object to those from the second object.

The PCS/ROOMS versions of these methods require that certain class variables be present in user-defined classes (this is similar to the idea of subclass responsibility in Smalltalk applications - see Chapter 2 for a more detailed discussion of these two approaches). These class variables are necessary to convey information about the instance variables to these methods. Display-list (used by the display-yourself method) is a list of all instance variables to be displayed, in the order in which they are to be displayed. Equality-list (used by the equal-to?, less-than?, and greater-than? methods) is a list of all instance variables to be compared, in the order in which

they are to be compared. And instvar-list (used by copy-from) is a list of all the instance variables that make up the object. Note that inherited instance variables must be included in these lists if they are to be displayed, compared, or copied by these methods.

The display-list and equality-list are actually made up of ordered pairs consisting of the variable name and the type of variable (variable, subobject, or dependent object). The instvar-list is made up of ordered triples consisting of the variable name, the type of variable, and (in the case of variables) if it is an atom or a list.

The Class Database

As discussed in Chapter 2, the class database was designed with a single instance variable, named members. This instance variable contains a list of all of the relations that are in the database, e.g., '(R S T) would indicate that the database has three relations: R, S, and T.

The database class is one of the two classes (the other class is the class relation) that do not need all of the previously discussed required methods. A display-yourself method is provided, but it is generally inappropriate to compare two databases. A copy-from method could be used to make a backup copy of the

database, but there are most likely more efficient ways of performing this function.

In PCS/ROOMS, a class variable (free-list) was also needed. In PC Scheme, a variable is considered local unless it is defined through the top-level Read-Eval-Print (REP) loop. As such, any variable defined within a method (or procedure) exists only as long as that method is executing.

The PCS/ROOMS record structure consists mainly of subobjects, so any relation operation (such as Union) with a permanent relation as its result needs to make copies of all of the source records (objects) that will be in the result. However, any new objects created by a method will not exist after the method is finished! Therefore, "enough" new variables (methods are provided in the class relation to determine how many) must be created ahead of time. Methods (such as Union) may then get one of these new variables and instantiate it as an object, and it will still exist after the method has finished.

PCS/ROOMS also provides a macro (make-generic-variables) which makes generic variables and adds them to the free-list variable (of the class database). This macro must be run at the top-level REP loop. The number of variables to be created must be specified, and the

macro must be run before any relation operation (such as Union) can be run.

The Class Relation

The class relation is the other class that does not need all of the previously discussed required methods. As discussed in Chapter 2, the class relation was designed with two instance variables - members and record-type. Members contains a list of all of the records in the relation, e.g., '(e1, e3, e5) would indicate that the records e1, e3, and e5 are in the relation. Record-type contains the (user-defined) class-name of the type of records that can be members of that particular relation. Both PCS/ROOMS and EOOPS/ROOMS define the class relation in this way.

Several methods are defined for the class relation. Display-yourself displays each record in the relation. Add-record and remove-record add and remove records to/from the relation, and get-records displays a list of all the records in the relation. These methods are essentially the same in both PCS/ROOMS and EOOPS/ROOMS.

The five basic relational operations (union, difference, selection, projection, and Cartesian product) are defined as methods of relations. Two additional operations, intersection and fast-intersection, are also defined.

The union, difference, selection, intersection, and fast-intersect methods have similar implementations in both PCS/ROOMS and EOOPS/ROOMS. They use the various equality methods to determine which records will be in the resulting relation.

The project and Cartesian-product methods use similar algorithms in both PCS/ROOMS and EOOPS/ROOMS. However, the EOOPS/ROOMS methods are shorter and simpler than those in PCS/ROOMS. This is partly because PC Scheme requires that new classes be defined at the top-level REP loop (similar to the way variables must be defined). Therefore, PCS/ROOMS' project and Cartesian-product are each split into two separate methods - one method to create the new class definition and one method to do the actual relational operation. The user, however, still need only send a single message to perform the operation - PCS/ROOMS uses its DBSEND macro to trap these messages and expand them into the appropriate new messages. The proper metaclass in EOOPS (see Chapter 3) avoids this problem.

Using multiple inheritance to build the records in EOOPS/ROOMS helped to simplify all of the methods which implement the relational operations. The methods (including those which are defined for the class record) simply worked their way through the hierarchy of the record rather than needing additional information about

the record structure to access each subobject (or dependent object) which makes up the record in PCS/ROOMS.

The Class Record

As discussed in Chapter 2, the class record was designed with no class variables or instance variables. Its only purpose is to provide methods to user-defined records, and as such can be considered to be an abstract class.

In PCS/ROOMS, the class record is defined as a regular class, and then treated as an abstract class. However, if EOOPS/ROOMS were simply to define a class record and then use that class as a superclass for user-defined records, the methods defined for the class record would not be able to access any of the variables defined for the user-defined record class.

These methods could have been defined in EOOPS/ ROOMS as generic methods and then instantiated by user-defined record classes, but that does not convey the "feeling" that these user-defined records are all of the same basic type. Therefore, the class record is defined as a generic class, which is then included (instantiated) in the user-defined record classes.

All of the previously discussed required methods are needed by the class record. The display-yourself,

equal-to?, greater-than?, less-than?, and copy-from methods are all instantiations of the generic methods which are provided in EOOPS/ROOMS, and inherited from the class named generic in PCS/ROOMS.

A meet-select-criteria? method is also provided. It determines if a record meets the selection criteria set forth by the select method of the class relation.

The algorithms used in both PCS/ROOMS and EOOPS/ROOMS for all of these methods are similar, except that EOOPS/ROOMS is able to work with inherited methods (since multiple inheritance is used to build the records) while PCS/ROOMS must work its way through the objects which make up the record.

User-Defined Record Classes

Since ROOMS is a relational system, the objects must be stored in records. In order to work properly with the class relation, user-defined record classes must include all of the methods which are provided by the class record.

The record structure in PCS/ROOMS is designed such that all the fields in the record are defined as instance variables. The instance variables can be simple variables (integers, for example), subobjects, or dependent objects. These variables may be inherited,

but the class designer is then responsible for ensuring that there are no name conflicts.

EOOPS/ROOMS, on the other hand, defines the record structure using multiple inheritance - the user simply defines the record by including each of the fields that make up the record as a superclass. Since variable name conflicts are not a problem in EOOPS (see Chapter 3), the user need only be concerned with the external interface of each of the inherited fields. This leads to one of the biggest benefits to a user of EOOPS/ROOMS over PCS/ROOMS - in order to send a message to one of the fields that make up the record, the message is simply sent to the record itself. In PCS/ROOMS, the message must be directed towards a particular part of the record (i.e., towards a particular subobject or dependent object, or towards a particular subobject or dependent object of a particular subobject or dependent object, etc.).

The record structure used in EOOPS/ROOMS could be modified to include simple variables, subobjects, and dependent objects (as in PCS/ROOMS) in addition to multiple inheritance. Doing so, however, would cause the benefits attributed to the simplified record structure using multiple inheritance to be lost.

User-Defined Classes

As previously discussed, the only limits on user-defined classes in ROOMS are the user's imagination and, of course, any limitations imposed by the host language and the hardware that is being used. This is another area in which the benefits of EOOPS really come into play as far as users of ROOMS are concerned, as they need not worry about what variables are inherited and what methods access them - each class is defined as an encapsulated unit which only needs to know about the external interface of its superclasses (see Chapter 3 for a more detailed discussion of this).

Once a user-defined class has been implemented, objects of that type can be stored in the database, but only if the user-defined classes include certain methods that are required by ROOMS. The previously discussed required methods (display-yourself, equal-to?, greater-than?, less-than?, and copy-from) are required for all user-defined classes to work properly within the database. These generic or default methods simply operate upon all of the variables and superclasses that make up a user-defined class. They may not meet the needs of all user-defined classes, but they do provide a starting point.

Conclusions

The two implementations of ROOMS have some similarities. This is due in part to the fact that they are both Lisp-based implementations. However, it is mainly because ROOMS was designed (as discussed in Chapter 2) using a fairly simple form of class definitions, along with their variables and methods, and every attempt was made to ensure that the basic design of ROOMS was not in any way language-dependent.

One of the differences between PCS/ROOMS and EOOPS/ROOMS is due to the way that global variables are needed for defining instances in PCS/ROOMS. This could become a problem in EOOPS/ROOMS if the host language for EOOPS should also have this problem.

The real differences between PCS/ROOMS and EOOPS/ROOMS, however, are the record structures used and the relational operations. Using subobjects and dependent objects in PCS/ROOMS led to a much more complex set of relational methods and default methods. Using multiple inheritance in EOOPS/ROOMS greatly simplified these methods, by simply accessing the appropriate method from each of the superclasses. Messages to the fields of an EOOPS/ROOMS record are simpler too, as they are sent to the record rather than to a specific part of the record. EOOPS/ROOMS has the added benefit in that it could be modified to allow the record structure to consist of a

combination of multiple inheritance, subobjects, and dependent objects, if this should be desired by the user.

More on these differences, along with various other concerns and suggestions for future research, are presented in the next chapter.

CHAPTER 5

SUMMARY, CONCLUSIONS, AND SUGGESTIONS FOR FUTURE RESEARCH

The goals of this research effort were two-fold. First was to design a Relational Object-Oriented Management System (ROOMS). ROOMS was designed so that it can be implemented in virtually any object-oriented programming language or added to any commercially available object-oriented database management system. Second was to design an Encapsulated Object-Oriented Programming System (EOOPS) to greatly simplify the implementation of any large-scale object-oriented project, including ROOMS.

In this chapter, we briefly summarize both ROOMS and EOOPS, present our conclusions, and offer suggestions for further research.

Summary

A detailed survey of the literature was accomplished. We investigated object-oriented programming in general, object-oriented database systems, genericity in existing languages (both conventional and object-oriented), and the relational

data model and object-oriented (or abstract data type) extensions to it.

ROOMS was designed in a language-independent manner, using only a general form of class and method definitions from object-oriented programming. This allows ROOMS to be implemented in virtually any object-oriented language, including PC Scheme and EOOPS.

Several problems were encountered in developing ROOMS, with most of the problems being directly related to the way that inheritance has been implemented in the available OOP languages. Various solutions were found for these problems so that ROOMS could be implemented in an available OOP language (and it was implemented in PC Scheme), but we believe that there is a better way - and that better way is EOOPS.

Not only is EOOPS better for applications such as ROOMS, the encapsulated inheritance scheme used in EOOPS should greatly simplify the development and long-term maintenance of any object-oriented programming project. EOOPS also includes features to define generic classes, which allows for properties to be inherited and redefined (in a very similar manner in which inheritance has been implemented in existing OOP languages). Generic methods have also been included to allow for even more sharing of similar methods between various classes.

Conclusions

We believe that the relational database model is a viable way to organize objects, but that it is not necessarily the best way for all applications. If ROOMS, which is based on the relational model, is included as part of an object-oriented database system, then there is no reason why a single database system cannot serve all of the data needs of a company. And by including ROOMS as part of an object-oriented database system, users need not be restricted to only relational access or non-relational access to the data.

ROOMS could also be implemented in any OOP language with appropriate I/O capabilities. This means that ROOMS could serve the needs of users who do not yet have access to a commercial object-oriented database system.

EOOPS preserves encapsulation, which should simplify the design and maintenance of any complex system. It will no longer be necessary to check all of the ancestor classes to avoid accidental name conflicts in the design process. It will also not be necessary to check all of the descendant classes when an existing class definition is modified (as long as the original external interface of the class is preserved).

A formal metaclass is included in EOOPS. While this may not mean much to the typical end-user, it does

allow system designers to test and implement future ideas affecting class definitions and inheritance.

Two forms of genericity were also included in EOOPS. Generic methods allow for operations such as display-yourself to be defined once and then instantiated as needed for various classes. Generic classes allow variables and methods to be inherited and redefined in essentially the same way that these features are handled in existing OOP languages.

Generic classes, however, have not been included to simply allow a non-encapsulated form of inheritance. They exist so that the similar portions of various classes can be made generic and shared between classes rather than creating possibly unwanted dependencies when defining one class based upon another.

Suggestions for Future Research

Having completed the design of both ROOMS and EOOPS, the development of production systems can now begin. There are, however, a few more questions that should be addressed first.

Future ROOMS Research. First and foremost, PCS/ROOMS is unacceptably slow. PC Scheme served well in developing a "proof of concept" system, but may not be acceptable in developing a production system. A faster or more efficient language and/or machine should be used.

In ROOMS, equal-to?, greater-than?, and less-than? methods are required, but only the equal-to? method has been provided so far. The greater-than? and less-than? methods should be fully designed and tested for inclusion in any future version of ROOMS.

The record structure to be used is still a questionable area. In PCS/ROOMS, records consist of variables, subobjects, and dependent objects. In EOOPS/ROOMS, records are built using multiple inheritance. Further research and experimentation may reveal what type of record structure is the best answer, or whether all of these possibilities should be included. The answer may partially lie in how objects are to be used or treated - do we need to be able to create an object, address it individually, and then attach it to one or more records? Or do we only need to be able to define a record and the fields (objects) which it consists of?

All of the records in a relation in ROOMS must be of the same type (i.e., instances of the same class). However, a relation is simply a collection of records, and it could be worthwhile to define another type of collection that is similar to a relation, but allow it to contain records of different types.

If ROOMS is to serve as the core of a new object-oriented database system rather than be added to an

existing system, then a proper data definition language (DDL) is required. This DDL must be capable of managing the various class definitions and their associated methods.

Whether ROOMS is added to an existing object-oriented database management system or developed as a standalone system, an external interface may be of great value to users that are accustomed to conventional relational database systems. An extension to SQL (similar to those extensions which have recently been designed for existing object-oriented systems) could serve as the basis for such an interface. The various SQL commands and extended-SQL commands would then send the appropriate messages to the various relations and records.

Future EOOPS Research. Since EOOPS has been designed but not implemented, the implementation of EOOPS is obviously the next logical step in its development. While we are very comfortable with the present definition of EOOPS, there are a few areas which could benefit from further research.

The keyword "send" has been included as a part of the message passing format, but most other object-oriented languages do not use this format. However, using "send" does seem to make it easier to pick out the

messages when scanning a lengthy program listing. It also allows the user to define other procedures to trap other commands (such as "dbsend") to perform some additional commands before or after "sending" the message. Further research and experimentation may help reveal whether the "send" should be required (as is), eliminated, or optional.

We have also discovered two possibilities for implementing class variables. Should a subclass receive a copy of an inherited class variable? Or should it receive a pointer to the class variable defined in the ancestor's class?

If a copy of an inherited class variable is used, then changes to the ancestor's variable has no effect on the subclass' variable, and vice versa. If a pointer to the ancestor's variable is used, then the appropriate message to any instance (object) in the hierarchy would change the value of that class variable for all other instances in the hierarchy. It would also be possible to allow both types of class variables.

Concluding Comments

This research effort resulted in the design and "proof of concept" of the Relational Object-Oriented Management System and the Encapsulated Object-Oriented Programming System. Acting on the suggestions for

future research should allow both ROOMS and EOOPS to make the transition from prototype systems to the status of commercially viable systems.

APPENDIX A

THE EOOPS LANGUAGE REFERENCE MANUAL

THE EOOPS LANGUAGE REFERENCE MANUAL

INTRODUCTION

The Encapsulated Object-Oriented Programming System (EOOPS) is an object-oriented extension for conventional programming languages, in much the same way that SCOOPS has been added to PC Scheme [Texas Instruments Inc. 1987b]. Many of the commands are modeled after SCOOPS commands, although it is not intended that EOOPS could only be added to PC Scheme. Most of the examples are given in a PC Scheme (Lisp) type of code, but it is also not intended that EOOPS could only be implemented in a Lisp-based language. One should also keep in mind that EOOPS is not a complete language - it is only an extension to a conventional language.

OVERVIEW

All of the entities in a typical object-oriented programming environment are called objects. However, since EOOPS is designed as an extension to a conventional language, any data type allowed by the host language may still be used.

An EOOPS object consists of variables, which determine the local state (or value) of the object.

Methods are defined for the object, and represent the only operations that the object knows how to perform. A message is sent to the object to tell it to perform one of its methods.

CLASSES AND GENERIC CLASSES

A class contains the description of one or more similar objects. An object is an instance of a class. The class definition consists of class variables, instance variables, superclasses, and instantiations of generic classes.

Class variables are shared in both name and value by all instances of the class - changing the value of a class variable for a single object changes the value of that class variable for all instances of the class. The instance variables are shared in name only by all instances of the class - each object has its own private set of the instance variables which are defined for the class (see the section on variables for more details on class and instance variables).

Methods (or procedures) are the only operations that instances of a class know how to perform (see the section on methods for more information). A class inherits the variables and methods of other classes that are specified as superclasses for the new class. This is an encapsulated form of inheritance, which is

different from the inheritance offered in "conventional" object-oriented languages (see the section on inheritance for more details).

A class may also instantiate generic classes. A generic class is similar to a regular (i.e., not generic) class, except that objects may not be instantiated for the generic class. All of the class variables, instance variables, and methods defined for the generic class are treated as though they were defined locally for the class instantiating the generic class. Instantiating a generic class is essentially the same as inheriting a class in conventional object-oriented languages. The format for a generic class definition is identical to that of a class definition, except that the definition begins with define-generic-class rather than define-class.

The format for a class definition is as follows:

```
(DEFINE-CLASS <class-name>
  [ (INSTANTIATE-GENERIC-CLASS
      { <class-name> }
  ) ]
  [ (CLASS-VARIABLES
      { ( <variable-name>     [initial-value] ) }
      [OPTIONS [GETTABLE] [SETTABLE] ]
  ) ]
  [ (INSTANCE-VARIABLES
      { ( <variable-name>     [initial-value] ) }
      [OPTIONS [GETTABLE] [SETTABLE] [INITTABLE] ]
  ) ]
  [ (SUPERCLASSES
      { <class-name> }
  ) ]
)
```

The options statement will cause certain methods to be automatically generated by the system for either the class variables or the instance variables. If gettable is specified, each variable will have a get-variable method generated which will return the value of that variable. If settable is specified, each variable will have a set-variable method generated which will set the value of that variable to the value specified in the message. If inittable (an option for instance variables only) is specified, then the variables may be initialized when an object is created.

The following example is the class definition for a class called widget:

```
(DEFINE-CLASS widget
  (CLASS-VARIABLES
    (widget-class-number  103)
  )
  (INSTANCE-VARIABLES
    (id-number  0)
    (color)
    OPTIONS GETTABLE SETTABLE
  )
)
```

Each instance of the class widget has a class variable called widget-class-number, which is initialized to the value 103. They also have two instance-variables, one called id-number (which is initialized to the value 0) and the other called color (which is not initialized to any value). Since the gettable and settable options are specified for instance

variables, the methods get-id-number, set-id-number, get-color, and set-color are created for the class description. The class widget has no superclasses. See the sections on variables and inheritance for more specific information on these subjects.

METHODS

Methods are defined for a class by using the DEFINE-METHOD command. The set of methods available to the instances of a class is called the external interface of the class. The set of methods available to the methods of a class is called the internal interface of the class. The external interface is a subset of the internal interface.

The difference between these two interfaces is mainly because all inherited methods are available to the methods of a class, but an inherited method is only available to the instances of the class if there is no new method with the same name defined for the class as the inherited method (see the section on inheritance for more information on this). Methods can also be specifically excluded from the external interface by using the external-delete-method command (to be defined shortly).

Note that the define-method operation is actually a method provided by EOOPS for each user-defined class,

therefore the command is simply a message which is sent to a class telling it to define a new method. The format of the define-method command is as follows:

```
(SEND <class-name> DEFINE-METHOD <method-name>
      ( {parameter} )
    {command}
)
```

A message telling an object to perform one of its methods is of the following form:

```
(SEND <object-name> <method-name> {parameter} )
```

Following are two methods defined for the previously defined class widget:

```
(send widget define-method set-id-and-color
      (new-id new-color)
  (set! id-number new-id)
  (set! color new-color)
)

(send widget define-method widget-display ()
 (let ((return-message nil))
  (set! return-message
    (append '(widget-number is)
            widget-class-number
            '(-)
            id-number
            '(, color is)
            color))
  return-message
))
```

The first method, set-id-and-color, simply sets the instance variables id-number and color to the appropriate values specified by the parameters new-id and new-color. The second method, widget-display, has no parameters. It returns the value of the class variable widget-class-number concatenated with the

value of the instance variable id-number, along with the value of the instance variable color.

A message is sent to an instance of the class widget to tell it to perform one of its methods. If object w is an instance of widget, then the message:

```
(send w set-id-and-color 15 'blue)
```

would cause the instance variable id-number of w to be set to 15 and the instance variable color to be set to blue. The message:

```
(send w widget-display)
```

would then return the following:

```
(widget-number is 103-15, color is blue)
```

The delete-method operation is another method which is provided for each user-defined class. The message:

```
(SEND <class-name> DELETE-METHOD <method-name> )
```

will cause method-name to be deleted from the available methods (i.e., from both the internal interface and the external interface) for class-name. Methods may also be deleted from only the external interface of a class by using the following message:

```
(SEND <class-name> EXTERNAL-DELETE-METHOD
      <method-name> )
```

See the section on inheritance for more information on this subject.

The make-instance method is also provided by the system for each user-defined class. The make-instance

message is sent to a user-defined class to create an instance of that class (i.e., to create an object that is a member of that class). The format of the make-instance message is:

```
(SEND <class-name> MAKE-INSTANCE <object-name>
      { ( <variable-name> value) }
)
```

This make-instance message will create object-name as an instance of class-name. If inittable has been specified in the options statement for the instance variables, then any of the variables along with an initial value may be included in the make-instance message. To make w an instance of the class widget, the following message would be used:

```
(send widget make-instance w)
```

SPECIAL METHODS

In addition to methods such as define-class and make-instance which are provided by EOOPS, other methods are also provided. These methods give the user information about various classes and objects.

Three such methods are get-superclasses, get-class-variables, and get-instance-variables. These methods return the appropriate information when sent to a (user-defined) class. The method get-superclasses returns a list of superclasses, while get-class-variables and get-instance-variables return a list of variables with their

initial values (if any were specified in the class definition).

For example, the message:

(send widget get-class-variables)

would return ((widget-class-number 103)), and the message:

(send widget get-instance-variables)

would return ((id-number 0) (color)), and the message:

(send widget get-superclasses)

would return nil, but if the class widget had any superclasses they would be returned as (superclass1, superclass2, etc.).

Another special method is class-of-object?. This method simply returns the class of an object, or nil if the "object" in question is not an object (for example, a simple variable would not be an object and this method would return nil).

GENERIC METHODS

Generic methods are defined by using the define-generic-method command, and instantiated by using the instantiate-generic-method command. When a generic method is defined and then instantiated for a class, it is treated the same as a regular (i.e., not generic) method defined for the class. It may be deleted as a method of the class by using the delete-method command,

or deleted from the external interface by using the external-delete-method command.

The format of the define-generic-method command is as follows:

```
(DEFINE-GENERIC-METHOD <method-name> ( {parameter} )
 ( {generic-parameter} )
  {command}
)
```

The format of the instantiate-generic-method command is then:

```
(SEND <class-name> INSTANTIATE-GENERIC-METHOD
  <method-name>    ( {parameter} )
  ( {generic-parameter} )
)
```

The number of parameters must be the same in both the definition and the instantiation of the generic method. They are input parameters to the method, and are matched up by name in the order they are given (i.e., if param1 is the first parameter in the define-generic-method command and p1 is the first parameter in the instantiate-generic-method, then p1 is matched up with param1).

Similarly, the number of generic-parameters must also be the same in both the definition and the instantiation of the generic method. These parameters can be constants, variables, and superclass names (something that is known in the class definition). They are also matched up by name in the order they are given.

Following is a generic method definition, which is then instantiated for the class widget:

```
(DEFINE-GENERIC-METHOD display-yourself
        (display-style) (display-list)
 (let ((return-message nil))
  (while (not (null? display-list))
    (if (equal? display-style 'titled)
      (set! return-message
        (append
          return-message
          (append
            (list (car display-list))
            '(:)
            (list (eval (car display-list)))))))
    (if (equal? display-style 'untitled)
      (set! return-message
        (append
          return-message
          (list (eval (car display-list))))))
    (set! display-list (cdr display-list)))
  return-message
))

(send widget instantiate-generic-method
      display-yourself (style) ('(id-number color))
)
```

The generic method display-yourself simply displays all of the variables in display-list. If display-style is the constant 'untitled, then the values of the variables are displayed. If display-style is the constant 'titled, then the variable names are also displayed.

In the instantiation of the generic method for the class widget, the input parameter style is matched up with the input parameter display-style in the method definition. The list '(id-number color) is matched up with the generic parameter display-list.

A display-yourself message can then be sent to an instance of the class widget. If object w is an instance of widget, with id-number set to 15 and color set to blue, then the message:

```
(send w display-yourself 'titled)
```

would return:

```
(id-number:15 color:blue)
```

and the message:

```
(send w display-yourself 'untitled)
```

would return:

```
(15 blue)
```

VARIABLES

Objects consist of class variables and instance variables. Class variables are shared in both name and value by all instances of the class. For example, the value of the class variable widget-class-number in the previously defined class widget is shared by all instances of that class. That is, if w1 and w2 are both instances of the class widget, then they share the variable widget-class-number; changing the value of this variable for one object implies that it has also changed for the other object.

Instance variables are not shared in value between different instances of the same class - each instance has its own set of instance variables. Changing the

value of an instance variable for one instance of a class has no effect on any other instance of that class.

The only way to access or modify the value of any variable is to send a message to an object telling it to perform a method which uses that variable. This is the form of encapsulation which is supported by other object-oriented systems. EOOPS takes encapsulation one step further through its approach to inheritance.

INHERITANCE

Inheritance implements a form of code sharing. When a superclass is specified, all of the class variables, instance variables, and methods of the superclass are inherited (including those which may have been inherited by the superclass itself).

In "conventional" object-oriented languages, the definitions of these inherited variables and methods can be changed (i.e., the variables and methods can be redefined) by using the same name over again in the new class definition. This approach makes the assumption that the designer of the class knows everything about every ancestor (i.e., a superclass, or an ancestor of a superclass) of the new class, and will choose new variable names and method names wisely to avoid accidentally redefining an inherited variable (or method). This approach becomes even more confusing when

multiple inheritance (allowing a class to have more than one superclass) is allowed (see the section on multiple inheritance).

EOOPS takes a more encapsulated approach. A class is defined to be everything that the superclass(es) is (are), plus the variables and methods which have been defined for the class. A set of variables is inherited from each superclass, along with a set of methods which represent the only way that these inherited variables can be accessed and manipulated.

Using the previously defined widget class, a two-tone-widget class can be defined with widget as a superclass as follows:

```
(DEFINE-CLASS two-tone-widget
  (INSTANCE-VARIABLES
    (color)
  )
  (SUPERCLASSES widget)
)
```

The class two-tone-widget can be thought of as "everything that a widget is, plus a new instance variable, color." This new instance variable color should not be confused with the one which is inherited. The inherited variables cannot be accessed by name, they can only be accessed by using the methods which were inherited with them. A set of variables, ((CLASS-VARIABLES widget-class-number), (INSTANCE-VARIABLES id-number, color)), has been inherited, along with the

methods set-id-number, set-color, get-id-number, get-color, set-id-and-color, and widget-display to access these variables. All of these inherited methods are available to each instance of the class two-tone-widget. Note that the class variable widget-class-number is initially set to 103, the same as for the class widget. However, since a new class has been defined, changing the value of widget-class-number for two-tone-widget has no effect on the value of widget-class-number for instances of the class widget.

New methods can be defined for the class two-tone-widget which directly access the instance variable color (the one defined for two-tone-widget, not the inherited one). These new methods can also use any of the inherited methods, but they cannot directly access any inherited variable. Following are two new methods for the class two-tone-widget:

```
(send two-tone-widget define-method
      set-id-and-color (new-id new-color)
  (set! color new-color)
  (send self widget.set-id-and-color new-id
        (concatenate 'light- new-color))
)

(send two-tone-widget define-method
      two-tone-widget-display
 (let ((return-message nil))
  (set! return-message
    (append (self widget.widget-display)
            '(,)
            color))
))
```

Note that the keyword self is simply a way of indicating that the message is to be sent to the same object that the original message was directed to.

The new method set-id-and-color replaces the set-id-and-color method which was inherited from the class widget. However, the old set-id-and-color method is still available to methods of the class two-tone-widget (as part of the internal interface) as widget.set-id-and-color. Note that the inherited method set-id-and-color is no longer available as part of the external interface for instances of the class two-tone-widget.

The new set-id-and-color method simply sets the instance variable color to the value specified as the new color, then sends itself the message:

```
(send self widget.set-id-and-color new-id
      "light-new-color")
```

which will set the inherited variables to the appropriate values. For example, if ww is an instance of the class two-tone-widget, then the message:

```
(send ww set-id-and-color 86 'blue)
```

This will set the instance variable color to blue and then send itself the message:

```
(send self widget.set-id-and-color 86 'light-blue)
```

which will set the inherited instance variable id-number to 86 and the inherited instance variable color to light-blue.

The method two-tone-widget-display will append the value of the instance variable color to the value returned by the message:

(send self widget.widget-display)

Note that the first widget in widget.widget-display is not required as there is no method name conflict (ambiguity) to resolve. However, it is recommended that the inherited method name be specified in this manner so that the routine can not be accidentally changed if a method widget-display were to be defined for the class two-tone-method. The message:

(send ww two-tone-widget-display)

would return the value:

(widget-number is 103-86, color is light-blue, blue)

The internal interface consists of the newly defined methods set-id-and-color and two-tone-widget-display, and the inherited methods set-id-number, set-color, get-id-number, get-color, and widget-display (all of which can also be referred to as widget.method-name), and the inherited method widget.set-id-and-color (which cannot be referred to simply as set-id-and-color due to a name conflict with the newly defined method with the same name).

The external interface consists of the newly defined methods set-id-and-color and two-tone-widget-display, and the inherited methods set-id-number, set-

color, get-id-number, get-color, and widget-display. The inherited method widget.set-id-and-color is not a part of the external interface and therefore cannot be directly accessed by any instance of the class two-tone-widget.

Note that the method widget.set-id-and-color is included in the internal interface, but not in the external interface. This is because the method set-id-and-color is defined (redefined) in the class two-tone-widget.

However, the designer of the class two-tone-widget may decide that having the methods set-color, get-color, and widget-display in the external interface may be confusing to users of the class. Therefore, these methods can be deleted from the external interface by sending the class two-tone-widget the following messages:

```
(send two-tone-widget external-delete-method
      set-color)
(send two-tone-widget external-delete-method
      get-color)
(send two-tone-widget external-delete-method
                  widget-display)
```

These methods are still in the internal interface of two-tone-widget (i.e., they can still be used by other methods defined for the class), but they are no longer defined for direct use by instances of the class (such as ww). Note that this mechanism can be used to give

the equivalent of private and public operations which are available in other languages such as C++ [Stroustrup 1986b].

MULTIPLE INHERITANCE

Multiple inheritance is simply allowing a class to inherit from two or more superclasses. However, method name conflicts become more of a problem. With single inheritance, the only possible method name conflict is when a method is defined for a class that has the same name as an inherited method. In this case the new method is part of both the external interface and the internal interface, and the inherited method is available only in the internal interface (as superclass-name.method-name).

With multiple inheritance, it is possible to inherit two (or more) methods with the same name. Both (or all) of these inherited methods are available as part of the internal interface (as superclass-name.method-name), but superclass-name's are not allowed in the external interface. A "first-come first-served" name conflict resolution scheme (similar to that used with "conventional" inheritance in languages such as PC Scheme) is used to determine which of the inherited methods will be in the external interface. Consider the following example:

```
(DEFINE-CLASS clock
  (INSTANCE-VARIABLES
    (color)
    (digital        TRUE)
    (alarm          TRUE)
    OPTIONS SETTABLE
  )
)
```

```
(DEFINE-CLASS radio
  (INSTANCE-VARIABLES
    (color)
    (fm-capability  TRUE)
    OPTIONS SETTABLE
  )
)
```

```
(DEFINE-CLASS clock-radio
  (INSTANCE-VARIABLES
    (battery-backup   TRUE)
    OPTIONS GETTABLE SETTABLE
  )
  (SUPERCLASSES clock radio)
)
```

The class clock-radio has a single instance variable, battery-backup, and two methods, get-battery-backup and set-battery-backup. It also has two superclasses, clock and radio. A set of instance variables has been inherited from each superclass, along with a set of methods to access those variables. So, the set of variables {(INSTANCE-VARIABLES color, digital, alarm)} and the methods set-color, set-digital, and set-alarm have been inherited from the class clock. And, the set of variables {(INSTANCE-VARIABLES color, fm-capability)} along with the methods set-color and set-fm-capability have been inherited from the class radio.

Note that two methods named set-color have been inherited. Both are in the internal interface (as clock.set-color and radio.set-color), but only one set-color will be in the external interface, and it will be

the clock.set-color method since clock is listed as a superclass before radio. If this is not what is desired for the class clock-radio, then the method set-color can be deleted from the external interface or a new method set-color can be defined for the class clock-radio.

Multiple inheritance from a single class is also available in EOOPS. Consider the following example:

```
(DEFINE-CLASS circle
  (INSTANCE-VARIABLES
    (center)
    (radius)
    (color)
    OPTIONS SETTABLE
  )
)

(DEFINE-CLASS target
  (SUPERCLASSES circle circle circle)
)
```

The class target inherits from the class circle three times, so it inherits three sets of variables {(INSTANCE VARIABLES center, radius, color)} and three sets of methods, set-center, set-radius, and set-color. To differentiate between the inherited methods, they are referred to as circle.1.set-center, circle.2.set-center, etc. The following methods for the class target demonstrate how these inherited methods might be used:

```
(send target define-method set-center (center)
  (send circle.1.set-center center)
  (send circle.2.set-center center)
  (send circle.3.set-center center)
)
```

```
(send target define-method set-radius (r1 r2 r3)
  (send circle.1.set-radius r1)
  (send circle.2.set-radius r2)
  (send circle.3.set-radius r3)
)

(send target define-method set-alternating-colors
      (c1 c2)
  (send circle.1.set-color c1)
  (send circle.2.set-color c2)
  (send circle.3.set-color c1)
)
```

METACLASS

The metaclass is a special class defined by EOOPS along with the define-class method. The metaclass is also considered to be an object. A user-defined class is a proper object in EOOPS since it is an instance of the metaclass. Note that the define-class message is assumed to be directed to the metaclass, so the following messages:

```
(send metaclass define-class ...)

(define-class ...)
```

are equivalent. Similarly, the define-generic-method message is also assumed to be directed to the metaclass.

User-defined methods can be added to the metaclass by using the define-method operation. The format of this method is the same as that used for adding methods to user-defined classes, i.e., the message:

```
(send metaclass define-method my-method (parameters)
      ...)
```

would cause my-method to be added to the metaclass, and

the message:

(send metaclass delete-method my-method)

would cause my-method to be deleted from the metaclass.

APPENDIX B

THE SOURCE CODE FOR THE IMPLEMENTATION OF ROOMS IN PC SCHEME

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ROOMS-00.PCS - section 00 of ROOMS in PC Scheme
; - this section contains the descriptions for
;   - OBJECT
;   - GENERIC
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; OBJECT
; - Description
;    - intended to be the root of every class hierarchy
;    - provides a "self" pointer since PCS does not
; - Superclasses
;    - none
; - Class Variables
;    - none
; - Instance Variables
;    - SELF
; - Methods
;    - (send-if-handles x IS-OBJECT?)
;      - #T if x is of type object, #F otherwise
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-class object
  (instvars   (self nil))
  (options gettable-variables settable-variables
           inittable-variables)
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; OBJECT IS-OBJECT?
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (object is-object?) () #T)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; OBJECT COMPILE-CLASS OBJECT
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(compile-class object)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; GENERIC
; - Description
;   - provides "generic" methods for user-defined
;     classes
; - Superclasses
;   - none
; - Class Variables
;   - none
; - Instance Variables
;   - none
; - Methods
;   - (send-if-handles x IS-GENERIC?)
;     - #T if x is of type generic, #F otherwise
;   - (send x COUNT-SUBOBJECTS)
;     - returns a count of subobjects used in the object
;     - recursive, (i.e., count subobjects of subobjects
;   - (send x COPY-FROM other)
;     - makes x a "copy" of other
;   - (send x DISPLAY-YOURSELF)
;     - "generic" display routine
;   - (send x EQUAL-TO? y)
;     - "generic" equal-to? routine, return #T or #F
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; GENERIC class definition
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-class generic
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; GENERIC IS-GENERIC?
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (generic is-generic?) () #T)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; GENERIC COPY-FROM
; - for each var in instvar-list (defined for each
;   individual class),
;   - if var is an instvar,
;     set var of new object equal to var of other
;   - if var is a subobject,
;     - get a new generic variable
;     - set var of new object to the generic variable
;     - make the generic variable an instance of the
;       class of other's var
;     - (send generic-variable COPY-FROM other's var)
;   - otherwise, set var of new object to nil
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (generic copy-from) (other)
 (let ((temp-instvar-list nil)   (next-var nil))
  (set! temp-instvar-list instvar-list)
  (if (or (equal? (class-of-object (eval self))
                  (class-of-object other))
          (and (send-if-handles (eval self) is-record?)
               (send-if-handles other is-record?)))
    (while (not (null? temp-instvar-list))
      (set! next-var (car temp-instvar-list))
      (cond
       ((equal? (cadr next-var) 'instvar)
        (if (not (equal? (car next-var) 'self))
          (eval '(send (eval self)
                       ,(concat 'set- (car next-var))
                  (send ,(eval other)
                    ,(concat 'get- (car next-var))))))))
       ((equal? (cadr next-var) 'subobject)
        (fluid-let ((tvar (send (eval rooms) get-var)))
         (eval '(set! ,(car next-var) (fluid tvar)))
         (eval '(set! ,(fluid tvar)
                  (make-instance ,(caddr next-var))))
         (send (eval (fluid tvar)) set-self
               (fluid tvar))
         (send (eval (fluid  tvar)) copy-from
               (eval (eval '(send ,other
                                  ,(concat 'get-
                                      (car next-var))))))))
       (#T
        (eval '(set! ,(car next-var) nil)))
      )
      (set! temp-instvar-list (cdr temp-instvar-list))
    )
  )
  (eval self)
))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; GENERIC COUNT-SUBOBJECTS
; - for each var in instvar-list (defined for each
;   individual class),
;   - if var is a subobject,
;     - add 1 to count
;     - add (send var COUNT-SUBOBJECTS) to count
;   - otherwise, do nothing
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (generic count-subobjects) ()
 (let ((cnt 0) (subcnt nil) (temp-instvar-list nil)
       (next-instvar nil))
  (set! temp-instvar-list instvar-list)
  (while (not (null? temp-instvar-list))
    (set! next-instvar (car temp-instvar-list))
    (cond
     ((equal? (cadr next-instvar) 'instvar)
      nil)
     ((equal? (cadr next-instvar) 'subobject)
      (set! cnt (+ cnt 1))
      (set! subcnt (eval '(send-if-handles
                           (eval ,(car next-instvar))
                           count-subobjects)))
      (if (not (null? subcnt))
        (set! cnt (+ cnt subcnt))))
     (#T
      nil)
    )
    (set! temp-instvar-list (cdr temp-instvar-list))
  )
  cnt
))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; GENERIC DISPLAY-YOURSELF
; - this version puts parens around subobjects
; - for each var in display-list (defined for each
;   individual class),
;   - if var is an instvar, append var to msg
;   - if var is a subobject, append
;     (send var display-yourself) to msg
;   - otherwise, append "?" to msg
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (generic display-yourself) ()
 (let ((msg nil)  (temp-display-list nil)
       (display-next nil))
  (set! temp-display-list display-list)
  (while (not (null? temp-display-list))
    (set! display-next (car temp-display-list))
    (cond
     ((equal? (cadr display-next) 'instvar)
      (set! msg
        (append msg
                (list (eval (car display-next))))))
     ((equal? (cadr display-next) 'subobject)
      (set! msg
        (append msg
                (list (eval '(send (eval
                                    ,(car display-next))
                                   display-yourself))))))
     (#T
      (set! msg (append msg '(?-) display-next '(-?))))
    )
    (set! temp-display-list (cdr temp-display-list))
  )
  msg
))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; GENERIC EQUAL-TO?
; - assume they are equal (i.e., set return-value to #T)
; - for each var in equality-list (defined for each
;   individual class),
;   - if var is an instvar, compare var to other's var
;   - if var is a subobject,
;     (send var EQUAL-TO? other's var)
;   - otherwise, set return-value to #F
;   - if return-value becomes #F, terminate "for" loop
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (generic equal-to?) (other)
 (let ((return-value #T)  (temp-equality-list nil)
       (check-next nil))
  (set! temp-equality-list equality-list)
  (if (equal? (class-of-object (eval self))
              (class-of-object other))
    (while (not (null? temp-equality-list))
      (set! check-next (car temp-equality-list))
      (cond
       ((equal? (cadr check-next) 'instvar)
        (if (not (equal?
          (eval (car check-next))
          (eval '(send ,other ,(concat 'get-
                                       (car check-next))))))
          (set! return-value #F)))
       ((equal? (cadr check-next) 'subobject)
        (if (not (eval '(send (eval ,(car check-next))
                              equal-to?
                              ,(eval '(send ,other
                                 ,(concat 'get-
                                 (car check-next)))))))
          (set! return-value #F)))
       (#T
        (set! return-value #F))
      )
      (if (null? return-value)
        (set! temp-equality-list nil))
     (set! temp-equality-list (cdr temp-equality-list))
    )
    (set! return-value #F))
  return-value
))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; COMPILE-CLASS GENERIC
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(compile-class generic)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; MACROS & FUNCTIONS TO CREATE OBJECTS & "NULL" OBJECTS
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(macro create-object (lambda (e)
  '(make-instance object 'self ',(cadr e))))

(macro create-generic (lambda (e)
  '(make-instance generic 'self ',(cadr e))))

(define null-object  (create-object  null-object))

(define null-generic (create-generic null-generic))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ROOMS-01.PCS - section 01 of ROOMS in PC Scheme
; - this section contains the descriptions for all
;   database "overhead" classes
;   - DATABASE, RELATION, & RECORD
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; DATABASE
; - Description
;   - a collection of relations
; - Superclasses
;   - OBJECT
; - Class Variables
;   - FREE-LIST
; - Instance Variables
;   - MEMBERS
; - Methods
;   - (send-if-handles x IS-DATABASE?)
;     - #T if x is of type database, #F otherwise
;   - (send x DISPLAY-YOURSELF)
;     - displays the names of relations in the database
;   - (send x GET-VAR)
;     - returns the first variable in FREE-LIST and
;       sets FREE-LIST to (cdr FREE-LIST)
;   - (send x ADD-MEMBER member-to-add)
;     - adds member-to-add to the member list if it is
;       of type relation
;   - (send x REMOVE-MEMBER member-to-remove)
;     - removes member-to-remove from the member list
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-class database
  (mixins object)
  (classvars (free-list nil))
  (instvars  (members nil))
  (options gettable-variables settable-variables
           inittable-variables)
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; DATABASE IS-DATABASE?
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (database is-database?) () #T)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; DATABASE DISPLAY-YOURSELF
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (database display-yourself) ()
  members
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; DATABASE GET-VAR
; - return the first generic variable in free-list and
;   set free-list to (cdr free-list)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (database get-var) ()
 (let ((var-to-return nil))
  (set! var-to-return (car free-list))
  (set! free-list (cdr free-list))
  var-to-return
))



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; DATABASE ADD-MEMBER
; - if member-to-add is of type relation and is not
;   already in the members list, then add it
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (database add-member) (member-to-add)
  (if (send-if-handles (eval member-to-add)
        is-relation?)
    (if (not (member member-to-add members))
      (set! members (append members
                            (list member-to-add)))
      members)
    nil)
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; DATABASE REMOVE-MEMBER
; - delete member-to-remove from members list
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (database remove-member)
               (member-to-remove)
  (set! members (delete! member-to-remove members))
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; COMPILE-CLASS DATABASE
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(compile-class database)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; MAKE-GENERIC-VARIABLES
; - a macro, (make-generic-variables cnt)
;   - makes cnt gensym variables which are added to
;     free-list of the database that is bound to the
;     global variable ROOMS
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(macro make-generic-variables (lambda (e)
'(let ((count nil) (free-list nil))
  (set! count ,(cadr e))
  (set! free-list (send (eval ROOMS) get-free-list))
  (set! count (- count (length free-list)))
  (cond
   ((>? count 0)
    (while (>? count 0)
      (set! free-list (append (list (gensym))
                              free-list))
      (eval '(define ,(car free-list) nil))
      (set! count (- count 1))
    )
    (send (eval ROOMS) set-free-list free-list))
   (#T
    free-list)
  )
)))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; RELATION
; - Description
;   - a collection of records
; - Superclasses
;   - OBJECT
; - Class Variables
;   - none
; - Instance Variables
;   - MEMBERS
;     - a list of all the records in the relation,
;       ex, '(e1 e2 e4 e7)
;   - RECORD-TYPE
;     - the class name of records that are allowed in
;       this relation, ex, 'emp-rec
; - Methods
;   - (send-if-handles x IS-RELATION?)
;     - #T if x is of type relation, #F otherwise
;   - (send x COUNT-SUBOBJECTS)
;     - returns a count of the total number of
;       subobjects in the relation
;     - uses the null object "null-RECORD-TYPE"
;   - (send x DISPLAY-YOURSELF)
;     - displays each record in the relation
;   - (send x ADD-RECORD record-to-add)
;     - add the record record-to-add to the relation
;       (i.e., to the member list)
;     - record-to-add must be a symbol
;       - i.e., if e1 is the record to be added,
;         must use 'e1 or x where (eval x) = 'e1,
;         e1 will NOT work
;     - record-to-add must be of type RECORD-TYPE
;   - (send x GET-INSTVAR-LIST)
;     - returns instvar-list of the class RECORD-TYPE
;       - does (send null-RECORD-TYPE get-instvar-list)
;   - (send x GET-RECORDS)
;     - returns members, the list of records
;   - (send x REMOVE-RECORD record-to-remove)
;     - removes record-to-remove from the relation
;       (i.e., from the member list)
;   - (send x CARTESIAN-PRODUCT-COUNT-SUBOBJECTS other)
;     (send x DIFFERENCE-COUNT-SUBOBJECTS other)
;     (send x INTERSECT-COUNT-SUBOBJECTS other)
;     (send x PROJECT-COUNT-SUBOBJECTS)
;     (send x SELECT-COUNT-SUBOBJECTS)
;     (send x UNION-COUNT-SUBOBJECTS other)
;     - returns the maximum number of subobjects that
;       will be needed by the corresponding operation
;   - relational algebra operations
;     - for all operations
;       - if result is a relation
```

```
;          - for each record that is both x and other,
;            get a gensym variable, make it an instance
;            of the appropriate record type, and then
;            send the gensym variable the message
;            copy-from the record in the intersection
;          - return the display of result
;        - if result is nil
;          - build a temporary relation consisting of the
;            records that are in both x and other
;            (i.e., use shared records)
;          - return the display of this temporary
;            relation
;        - if result is a symbol (such as 'relation-x)
;          - build a temporary relation consisting of the
;            records that are in both x and other
;            (i.e., use shared records)
;          - set the self pointer of the temporary
;            relation to the given symbol
;          - return the temporary relation
;      - (send x CARTESIAN-PRODUCT other new-rec-class
;                new-relation)
;        - returns a sequence of commands which, when
;          executed,
;          - performs (make-generic-variables
;            cartesian-product-count-subobjects)
;          - defines & compiles the class new-rec-class,
;            based on the definition of x and other
;          - defines "new-relation" to be an instance of
;            relation
;          - performs (send x cartesian-product2 other
;                             new-relation)
;      - (send x CARTESIAN-PRODUCT2 other result)
;        - used by CARTESIAN-PRODUCT
;        - if the result's fields consist of x's fields
;          followed by other's fields then get a
;          generic-variable and call
;          (send generic-variable copies-from x-record
;          other-record) for each possible pair of x
;          records and other records
;      - (send x DIFFERENCE other result)
;        - make result the difference of x & other
;      - (send x FAST-INTERSECT other result)
;        - make result the intersection of x & other,
;          using the EQUAL-TO? method to determine which
;          records to pick
;      - (send x INTERSECT other result)
;        - make x the intersection of x & other,
;          using the "classic" formula
;          result = (x - (x - other)), implemented as
;          - temp-relation = (x - other)
;          - result = (x - temp-relation)
```

```
;      - (send x PROJECT (field-list) new-rec-class
;                                       new-relation)
;        - returns a sequence of commands which, when
;          executed,
;          - performs (make-generic-variables
;                       project-count-subobjects)
;          - defines & compiles the class new-rec-class,
;            based on the definition of the records that
;            may be added to x
;          - makes "new-relation" an instance of relation
;          - performs (send x project2 (field-list)
;                             new-relation)
;      - (send x PROJECT2 (field-list) result)
;        - used by project
;        - if the appropriate fields are the only fields
;          in records of the type allowed by the result
;          relation, then
;          - for each record in other's member list, get
;            a gensym variable, make it an instance of
;            the appropriate record type, and then send
;            the gensym variable the message copy-from
;          the record in other's member list
;      - (send x SELECT which op what)
;        - make result those records of x that meet the
;          selection criteria of which, op, & what
;      - (send x UNION other result)
;        - make result the union of x & other
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-class relation
  (mixins object)
  (instvars   (members nil)  (record-type nil))
  (options gettable-variables settable-variables
           inittable-variables)
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; RELATION IS-RELATION
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (relation is-relation?) () #T)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; RELATION COUNT-SUBOBJECTS
; - count the number of subobjects in a record of the
;   relation, and add 1 to it (for the record itself)
; - then multiply by number of records in the relation
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (relation count-subobjects) ()
  (if (null? members)
   0
   (* (length members)
      (+ 1 (send (eval (concat 'null- record-type))
                 count-subobjects))))
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; RELATION DISPLAY-YOURSELF
; - concatenate the results of
;   (send record DISPLAY-YOURSELF)
;   for each record in the relation
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (relation display-yourself) ()
 (let ((member-list members) (msg nil))
  (while (not (null? member-list))
    (set! msg
      (append msg
              (list (send (eval (car member-list))
                          display-yourself))))
    (set! member-list (cdr member-list))
  )
  (pprint msg)
))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; RELATION ADD-RECORD
; - if record-to-add is of the appropriate record type
;   and is not already a member of the relation, then
;   add it to the members list
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (relation add-record) (record-to-add)
  (if (and (symbol? record-to-add)
           (equal?
             (class-of-object (eval record-to-add))
             record-type))
    (if (not (member record-to-add members))
      (set! members
        (append members
                (eval '(list (quote ,record-to-add)))))
      members)
    nil)
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; RELATION GET-INSTVAR-LIST
; - returns instvar-list of the records in the relation
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (relation get-instvar-list) ()
 (let ((null-rec (concat 'null- record-type)))
  (send (eval null-rec) get-instvar-list)
))



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; RELATION GET-RECORDS
; - display the members of the relation
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (relation get-records) ()
  members
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; RELATION REMOVE-RECORD
; - remove record-to-remove from the members list
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (relation remove-record)
               (record-to-remove)
  (set! members (delete! record-to-remove members))
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; RELATION CARTESIAN-PRODUCT-COUNT-SUBOBJECTS
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (relation
               cartesian-product-count-subobjects)
               (other)
 (let ((my-rec-count (length members))
       (other-rec-count (length
                          (send other get-members)))
       (rec-count 0))
  (if (< my-rec-count other-rec-count)
    (set! rec-count my-rec-count)
    (set! rec-count other-rec-count))
  (+ (* rec-count (send (eval self) count-subobjects))
     (* rec-count (send other count-subobjects)))
))



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; RELATION DIFFERENCE-COUNT-SUBOBJECTS
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (relation difference-count-subobjects)
               (other)
  (send (eval self) count-subobjects)
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; RELATION INTERSECT-COUNT-SUBOBJECTS
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (relation intersect-count-subobjects)
               (other)
 (let ((my-cnt 0) (other-cnt 0))
  (set! my-cnt (send (eval self) count-subobjects))
  (set! other-cnt (send (eval other) count-subobjects))
  (if (< my-cnt other-cnt)
    my-cnt
    other-cnt)
))



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; RELATION PROJECT-COUNT-SUBOBJECTS
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (relation project-count-subobjects) ()
  (send (eval self) count-subobjects)
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; RELATION SELECT-COUNT-SUBOBJECTS
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (relation select-count-subobjects) ()
  (send (eval self) count-subobjects)
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; RELATION UNION-COUNT-SUBOBJECTS
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (relation union-count-subobjects) (other)
  (+ (send (eval self) count-subobjects)
     (send (eval other) count-subobjects))
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; RELATION CARTESIAN-PRODUCT
; - build the new record class, instantiate the new
;   relation, and send self the message to
;   cartesian-product2 (which does the actual
;   cartesian product operation)
;   - build gen-var-msg to create the appropriate number
;     of generic variables
;   - build define-record-msg to define new-rec-class
;     - instvar-list is the concatenation of mine and
;       cdr of other's instvar-list (to eliminate the
;       "extra" self variable)
;     - display-list is the concatenation of mine and
;       other's display-list
;     - equality-list is the concatenation of mine and
;       other's equality-list
;     - instvars are all of the variables from the new
;       instvar-list
;   - define-record-method-msg implements the
;     IS-"NEW-REC-CLASS" method
;   - compile-record-msg compiles the new record class
;   - make-null-record-msg makes the object
;     NULL-"NEW-REC-CLASS"
;   - make-relation-instance-msg makes new-relation an
;     instance of new-rec-class
;   - set-record-type-msg sends the new-relation the
;     message to set its record-type to new-rec-class
;   - send-db-add-member-msg sends db the message to add
;     the new-relation to its list of member relations
;   - send-cartesian-product2-msg sends the message to
;     go ahead with the cartesian product
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (relation cartesian-product)
               (other new-rec-class new-relation)
 (let ((my-instvar-list
         (send (eval (car members)) get-instvar-list))
       (other-instvar-list
         (send (eval (car (send other get-members)))
           get-instvar-list))
       (new-instvar-list nil)
       (working-instvar-list nil)
       (new-instvars nil)
       (my-display-list
         (send (eval (car members)) get-display-list))
       (other-display-list
         (send (eval (car (send other get-members)))
           get-display-list))
       (new-display-list nil)
       (my-equality-list
         (send (eval (car members)) get-equality-list))
```

```
      (other-equality-list
        (send (eval (car (send other get-members)))
          get-equality-list))
      (new-equality-list nil)
      (gen-var-msg nil)
      (define-record-msg nil)
      (define-record-method-msg nil)
      (compile-record-msg nil)
      (make-null-record-msg nil)
      (make-relation-instance-msg nil)
      (set-record-type-msg nil)
      (send-db-add-member-msg nil)
      (send-cartesian-product2-msg nil)
      (msg nil))
(set! gen-var-msg (list
  (append '(make-generic-variables)
          (list (send (eval self)
                      cartesian-product-count-subobjects
                      other)))))
(set! new-instvar-list
  (append my-instvar-list
          (cdr other-instvar-list)))
(set! new-display-list (append my-display-list
                               other-display-list))
(set! new-equality-list (append my-equality-list
                                other-equality-list))
(set! working-instvar-list (cdr new-instvar-list))
(while (not (null? working-instvar-list))
  (set! new-instvars
    (append new-instvars
            (list (append
              (list (caar working-instvar-list))
              (list '(quote
                ,(concat 'null-
                (caddar working-instvar-list)))))))))
  (set! working-instvar-list
    (cdr working-instvar-list)))
(set! define-record-msg
  (list
    (append
      '(define-class)
      (list new-rec-class)
      '((mixins record))
      (list
        (append
          '(classvars)
          (list
            (append
              '(instvar-list)
              (list '(quote ,new-instvar-list))))
          (list
```

```
                (append
                  '(display-list)
                  (list '(quote ,new-display-list))))
              (list
                (append
                  '(equality-list)
                  (list '(quote ,new-equality-list))))))
          (list (append '(instvars) new-instvars))
          '((options gettable-variables settable-variables
                     inittable-variables)))))
    (set! define-record-method-msg
      (list
        (eval '(list 'define-method
                 '(,new-rec-class
                    ,(concat
                       (concat 'is- new-rec-class) '?))
                    '() '#T))))
    (set! compile-record-msg (list
      (eval '(list 'compile-class ',new-rec-class))))
    (set! make-null-record-msg (list
      (eval '(list 'define ',(concat 'null- new-rec-class)
               '(make-instance ,new-rec-class (quote self)
                 (quote ,(concat 'null-
                                 new-rec-class)))))))
    (set! make-relation-instance-msg (list
      (eval '(list 'define ',new-relation
        '(make-instance relation (quote self)
          (quote ,new-relation))))))
    (set! set-record-type-msg (list
      (eval '(list 'send ',new-relation 'set-record-type
                         (quote ',new-rec-class)))))
    (set! send-db-add-member-msg (list
      (eval '(list 'send rooms 'add-member
                         (quote ',new-relation)))))
    (set! send-cartesian-product2-msg (list
      (eval '(list 'send ',self 'cartesian-product2 ,other
                         ',new-relation))))
    (set! msg (append
      '(begin)  gen-var-msg
      define-record-msg  define-record-method-msg
      compile-record-msg  make-null-record-msg
      make-relation-instance-msg  set-record-type-msg
      send-db-add-member-msg send-cartesian-product2-msg))
    msg
))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; CARTESIAN-PRODUCT2
; - does minimal error checking (placing result in
;   legit-cp) to determine if self and other are
;   appropriate for building result
; - for each record in self and each record in other,
;   - get a generic variable and make it an instance of
;     result's record type
;   - using COPIES-FROM, make this new record a copy of
;     the combination of self's and other's record
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (relation cartesian-product2)
               (other result)
 (let ((my-instvar-list
         (send (eval self) get-instvar-list))
       (others-instvar-list
         (send other get-instvar-list))
       (results-instvar-list
         (send result get-instvar-list))
       (legit-cp #T)
       (my-members members)
       (others-members (send other get-members))
       (results-members nil)
       (save-my-members nil)
       (save-others-members nil)
       (results-record-type
         (send result get-record-type)))
  (if (= (+ (length results-instvar-list) 1)
         (+ (length my-instvar-list)
            (length others-instvar-list)))
    (begin
      (while (not (null? my-instvar-list))
        (if (not (equal? (car my-instvar-list)
                         (car results-instvar-list)))
          (begin (set! legit-cp #F)
                 (set! my-instvar-list nil)
                 (set! results-instvar-list nil)))
        (set! my-instvar-list (cdr my-instvar-list))
        (set! results-instvar-list
          (cdr results-instvar-list)))
      (set! others-instvar-list
        (cdr others-instvar-list))
      (while (not (null? others-instvar-list))
        (if (not (equal? (car others-instvar-list)
                         (car results-instvar-list)))
          (begin (set! legit-cp #F)
                 (set! others-instvar-list nil)
                 (set! results-instvar-list nil)))
        (set! others-instvar-list
          (cdr others-instvar-list))
```

```
          (set! results-instvar-list
            (cdr results-instvar-list))))
      (set! legit-cp #F))
    (if legit-cp
      (begin
        (set! save-others-members others-members)
        (while (not (null? my-members))
          (while (not (null? others-members))
           (fluid-let ((tvar (send (eval rooms) get-var)))
            (eval '(set! ,(fluid tvar)
                     (make-instance ,results-record-type)))
            (send (eval (fluid tvar)) set-self
                  (fluid tvar))
            (send (eval (fluid tvar)) copies-from
                  (eval (car my-members))
                  (eval (car others-members)))
            (set! results-members (append
              results-members (list (fluid tvar))))
            (set! others-members (cdr others-members))))
          (set! others-members save-others-members)
          (set! my-members (cdr my-members)))
        (send result set-members results-members))
        (send result display-yourself))
))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; RELATION DIFFERENCE
; - build shared-result-members as those records that
;   are in the difference, using EQUAL-TO?
; - then
;   - if result is nil
;     - build & display temp-result, a relation with
;       shared-result-members
;   - if result is a symbol
;     - build & return temp-result, a relation with self
;       set to the given symbol
;   - if result is a relation
;     - copy each of the shared-result-members, placing
;       the new records in result-members
;     - send result the list of result-members
;     - send result the message display-yourself
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (relation difference) (other result)
 (let ((my-members members)  (other-members nil)
       (shared-result-members nil)  (result-members nil)
       (add-flag #T)  (temp-result nil))
  (cond
   ((and (equal? (class-of-object other) 'relation)
         (or (null? result)
             (symbol? result)
             (equal? (class-of-object result)
                     'relation)))
    (set! other-members (send other get-members))
    (while (not (null? my-members))
      (while (not (null? other-members))
        (cond
         ((send (eval (car my-members)) equal-to?
                (eval (car other-members)))
          (set! other-members nil)
          (set! add-flag nil))
         (#T
          (set! other-members (cdr other-members)))
      ))
      (if add-flag
        (set! shared-result-members
          (append shared-result-members
                  (list (car my-members))))
        (set! add-flag #T))
      (set! my-members (cdr my-members))
      (set! other-members (send other get-members))
    )
    (cond
     ((or (null? result)
          (symbol? result))
      (set! temp-result (make-instance relation))
```

```
        (send temp-result set-members
              shared-result-members)
        (if (symbol? result)
         (begin
          (send temp-result set-self result)
          temp-result)
         (begin
          (send temp-result set-self 'temp-result)
          (send temp-result display-yourself))
        ))
       (#T
        (while (not (null? shared-result-members))
         (fluid-let ((tvar (send (eval rooms) get-var)))
          (eval '(set! ,(fluid tvar)
                       (make-instance ,record-type)))
          (send (eval (fluid tvar)) set-self (fluid tvar))
          (send (eval (fluid tvar)) copy-from
            (eval '(eval ,(car shared-result-members))))
          (set! result-members (append
            result-members (list (fluid tvar))))
          (set! shared-result-members
            (cdr shared-result-members))
        ))
        (send result set-members result-members)
        (send result display-yourself))))
    (#T
     (list 'cannot 'difference 'unlike 'relations))
  )
))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; RELATION INTERSECT
; - use the "classic" formula for intersection,
;   formula (self - (self - other))
; - no need to worry about what result is as the
;   difference method will take care of that
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (relation intersect) (other result)
 (let ((temp-relation nil))
  (cond
   ((equal? (class-of-object other) 'relation)
    (set! temp-relation
      (send (eval self) difference other
            'temp-relation))
    (send (eval self) difference temp-relation result))
   (#T
    (list 'cannot 'intersect 'unlike 'relations))
  )
))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; RELATION FAST-INTERSECT
; - build shared-result-members as those records that
;   are in the intersection, using the EQUAL-TO? method
; - then
;   - if result is nil
;     - build & display temp-result, a relation with
;       shared-result-members
;   - if result is a symbol
;     - build & return temp-result, a relation with self
;       set to the given symbol
;   - if result is a relation
;     - copy each of the shared-result-members, placing
;       the new records in result-members
;     - send result the list of result-members
;     - send result the message display-yourself
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (relation fast-intersect) (other result)
 (let ((my-members members)  (other-members nil)
       (shared-result-members nil)  (result-members nil)
       (add-flag nil)  (temp-result nil))
  (cond
   ((and (equal? (class-of-object other) 'relation)
         (or (null? result)
             (symbol? result)
             (equal? (class-of-object result)
                     'relation)))
    (set! other-members (send other get-members))
    (while (not (null? other-members))
      (while (not (null? my-members))
        (cond
         ((send (eval (car my-members)) equal-to?
                (eval (car other-members)))
          (set! my-members nil)
          (set! add-flag #T))
         (#T
          (set! my-members (cdr my-members)))
      ))
      (cond
       (add-flag
        (set! shared-result-members
          (append shared-result-members
                  (list (car other-members))))
        (set! add-flag nil))
      )
      (set! my-members members)
      (set! other-members (cdr other-members))
    )
    (cond
     ((or (null? result)
```

```
          (symbol? result))
       (set! temp-result (make-instance relation))
       (send temp-result set-members
             shared-result-members)
       (if (symbol? result)
        (begin
         (send temp-result set-self result)
         temp-result)
        (begin
         (send temp-result set-self 'temp-result)
         (send temp-result display-yourself))
       ))
      (#T
       (while (not (null? shared-result-members))
        (fluid-let ((tvar (send (eval rooms) get-var)))
         (eval '(set! ,(fluid tvar)
                      (make-instance ,record-type)))
         (send (eval (fluid tvar)) set-self (fluid tvar))
         (send (eval (fluid tvar)) copy-from
               (eval '(eval
                       ,(car shared-result-members))))
         (set! result-members
           (append result-members (list (fluid tvar))))
         (set! shared-result-members
           (cdr shared-result-members))
       ))
       (send result set-members result-members)
       (send result display-yourself))))
   (#T
    (list 'cannot 'fast-intersect 'unlike 'relations))
  )
))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; RELATION PROJECT
; - build the new record class, instantiate the new
;   relation, and send self the message to project2
;   (which does the actual project operation)
;   - build gen-var-msg to create the appropriate number
;     of generic variables
;   - build define-record-msg to define new-rec-class
;     - instvar-list are those entries from my
;       instvar-list which are being projected
;     - display-list are those entries from my
;       instvar-list which are being projected
;     - equality-list are those entries from my
;       instvar-list which are being projected
;     - instvars are those variables which are being
;       projected
;   - define-record-method-msg implements the
;     IS-"NEW-REC-CLASS" method
;   - compile-record-msg compiles the new record class
;   - make-null-record-msg makes the object
;     NULL-"NEW-REC-CLASS"
;   - make-relation-instance-msg makes new-relation an
;     instance of new-rec-class
;   - set-record-type-msg sends the new-relation the
;     message to set its record-type to new-rec-class
;   - send-db-add-member-msg sends the db the message to
;     add the new-relation
;     to its list of member relations
;   - send-project2-msg sends the message to go ahead
;     with the project now that the result relation and
;     its record class have been defined
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (relation project)
               (field-list new-rec-class new-relation)
 (let ((working-field-list field-list)
       (instvar-list
         (send (eval (car members)) get-instvar-list))
       (working-instvar-list nil)
       (new-instvar-list '((self instvar aatom)))
       (display-list
         (send (eval (car members)) get-display-list))
       (working-display-list nil)
       (new-display-list nil)
       (equality-list
         (send (eval (car members)) get-equality-list))
       (working-equality-list nil)
       (new-equality-list nil)
       (new-instvars nil)
       (gen-var-msg nil)
       (define-record-msg nil)
```

```
      (define-record-method-msg nil)
      (compile-record-msg nil)
      (make-null-record-msg nil)
      (make-relation-instance-msg nil)
      (set-record-type-msg nil)
      (send-db-add-member-msg nil)
      (send-project2-msg nil)
      (msg nil))
(set! working-instvar-list instvar-list)
(set! working-display-list display-list)
(set! working-equality-list equality-list)
(set! gen-var-msg (list (append
  '(make-generic-variables)
  (list (send (eval self)
              project-count-subobjects)))))
(while (not (null? working-field-list))
  (while (not (null? working-instvar-list))
    (if (equal? (car working-field-list)
                (caar working-instvar-list))
      (begin
        (set! new-instvar-list (append
          new-instvar-list
          (list (car working-instvar-list))))
        (set! new-instvars
          (append
            new-instvars
            (list
              (append
                (list (car working-field-list))
                (list '(quote ,(concat
                  'null-
                  (caddar working-instvar-list))))))))
        (set! working-instvar-list nil))
      (set! working-instvar-list
        (cdr working-instvar-list))))
  (while (not (null? working-display-list))
    (if (equal? (car working-field-list)
                (caar working-display-list))
      (begin
        (set! new-display-list
          (append new-display-list
                  (list (car working-display-list))))
        (set! working-display-list nil))
      (set! working-display-list
        (cdr working-display-list))))
  (while (not (null? working-equality-list))
    (if (equal? (car working-field-list)
                (caar working-equality-list))
      (begin
        (set! new-equality-list
          (append new-equality-list
```

```
                    (list (car working-equality-list))))
          (set! working-equality-list nil))
        (set! working-equality-list
          (cdr working-equality-list))))
    (set! working-field-list (cdr working-field-list))
    (set! working-instvar-list instvar-list)
    (set! working-display-list display-list)
    (set! working-equality-list equality-list))
(set! define-record-msg (list (append
  '(define-class) (list new-rec-class)
  '((mixins record))
  (list (append
    '(classvars)
    (list (append '(instvar-list)
                  (list '(quote ,new-instvar-list))))
    (list (append '(display-list)
                  (list '(quote ,new-display-list))))
    (list (append '(equality-list)
                  (list '(quote ,new-equality-list))))))
    (list (append '(instvars) new-instvars))
    '((options gettable-variables settable-variables
               inittable-variables)))))
(set! define-record-method-msg (list (eval
  '(list 'define-method
     '(,new-rec-class
     ,(concat (concat 'is- new-rec-class) '?)) '()
     '#T))))
(set! compile-record-msg (list (eval
  '(list 'compile-class ',new-rec-class))))
(set! make-null-record-msg (list (eval
  '(list 'define ',(concat 'null- new-rec-class)
         '(make-instance ,new-rec-class (quote self)
           (quote ,(concat 'null- new-rec-class)))))))
(set! make-relation-instance-msg (list (eval
  '(list 'define ',new-relation
    '(make-instance relation (quote self)
      (quote ,new-relation))))))
(set! set-record-type-msg (list (eval
  '(list 'send ',new-relation 'set-record-type
    (quote ',new-rec-class)))))
(set! send-db-add-member-msg (list (eval
  '(list 'send rooms 'add-member
    (quote ',new-relation)))))
(set! send-project2-msg (list (eval
  '(list 'send ',self 'project2 (quote ',field-list)
               ',new-relation))))
(set! msg (append
  '(begin)  gen-var-msg  define-record-msg
  define-record-method-msg  compile-record-msg
  make-null-record-msg  make-relation-instance-msg
  set-record-type-msg  send-db-add-member-msg
```

```
    send-project2-msg))
  msg
))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; RELATION PROJECT2
; - does minimal error checking (placing result in
;   legit-project) to determine if self is appropriate
;   for building result
; - for each record in self,
;   - get a generic variable and make it an instance of
;     result's record type
;   - use COPY-FROM to make this new record a copy of
;     the appropriate fields from self's record
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (relation project2) (field-list result)
 (let ((results-rec-type nil)  (results-rec nil)
       (results-instvar-list nil)  (legit-project #T)
       (my-members members)  (result-members nil))
  (set! results-rec-type (send result get-record-type))
  (set! results-rec
    (eval '(make-instance ,results-rec-type)))
  (set! results-instvar-list
    (cdr (send results-rec get-instvar-list)))
  (if (= (length field-list)
         (length results-instvar-list))
    (while (not (null? field-list))
      (if (not (equal? (car field-list)
                       (caar results-instvar-list)))
        (begin (set! legit-project #F)
               (set! field-list nil)))
      (set! results-instvar-list
        (cdr results-instvar-list))
      (set! field-list (cdr field-list)))
    (set! legit-project #F))
  (cond
   (legit-project
    (while (not (null? my-members))
     (fluid-let ((tvar (send (eval rooms) get-var)))
      (eval '(set! ,(fluid tvar)
               (make-instance ,results-rec-type)))
      (send (eval (fluid tvar)) set-self (fluid tvar))
      (send (eval (fluid tvar)) copy-from
            (eval '(eval ,(car my-members))))
      (set! result-members (append result-members
                                   (list (fluid tvar))))
      (set! my-members (cdr my-members))
    ))
    (send result set-members result-members)
    (send result display-yourself))
  )
))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; RELATION SELECT
; - select records to build shared-result-members
; - then
;   - if result is nil
;     - build & display temp-result, a relation with
;       shared-result-members
;   - if result is a symbol
;     - build & return temp-result, a relation with self
;       set to the given symbol
;   - if result is a relation
;     - copy each of the shared-result-members, placing
;       the new records in result-members
;     - send result the list of result-members
;     - send result the message display-yourself
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (relation select) (which op what result)
 (let ((my-members members)  (shared-result-members nil)
       (result-members nil)  (temp-result nil))
  (cond
   ((or (null? result)
        (symbol? result)
        (equal? (class-of-object result) 'relation))
    (while (not (null? my-members))
      (if (send (eval (car my-members))
                meet-select-criteria? which op what)
        (set! shared-result-members
          (append
            shared-result-members
            (list (car my-members)))))
      (set! my-members (cdr my-members))
    )
    (cond
     ((or (null? result)
          (symbol? result))
      (set! temp-result (make-instance relation))
      (send temp-result set-members
            shared-result-members)
      (if (symbol? result)
       (begin
        (send temp-result set-self result)
        temp-result)
       (begin
        (send temp-result set-self 'temp-result)
        (send temp-result display-yourself))
      ))
     (#T
      (while (not (null? shared-result-members))
       (fluid-let ((tvar (send (eval rooms) get-var)))
        (eval '(set! ,(fluid tvar)
```

```
                    (make-instance ,record-type)))
              (send (eval (fluid tvar)) set-self (fluid tvar))
              (send (eval (fluid tvar)) copy-from
                    (eval '(eval
                      ,(car shared-result-members))))
              (set! result-members (append
                result-members (list (fluid tvar))))
              (set! shared-result-members
                (cdr shared-result-members))
            ))
            (send result set-members result-members)
            (send result display-yourself))))
        (#T
         (list ' improper 'result))
      )
    ))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; RELATION UNION
; - build shared-result-members as those records that
;   are in the union
; - then
;   - if result is nil
;     - build & display temp-result, a relation with
;       shared-result-members
;   - if result is a symbol
;     - build & return temp-result, a relation with self
;       set to the given symbol
;   - if result is a relation
;     - copy each of the shared-result-members, placing
;       the new records in result-members
;     - send result the list of result-members
;     - send result the message display-yourself
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (relation union) (other result)
 (let ((my-members members)  (other-members nil)
       (shared-result-members members)
       (result-members nil)  (add-flag #T)
       (temp-result nil))
  (cond
   ((and (equal? (class-of-object other) 'relation)
         (or (null? result)
             (symbol? result)
             (equal? (class-of-object result)
                     'relation)))
    (set! other-members (send other get-members))
    (while (not (null? other-members))
      (while (not (null? my-members))
        (cond
         ((send (eval (car my-members)) equal-to?
                (eval (car other-members)))
          (set! my-members nil)
          (set! add-flag nil))
         (#T
          (set! my-members (cdr my-members)))
      ))
      (if add-flag
        (set! shared-result-members (append
          shared-result-members
          (list (car other-members))))
        (set! add-flag #T))
      (set! my-members members)
      (set! other-members (cdr other-members))
    )
    (cond
     ((or (null? result)
          (symbol? result))
```

```
      (set! temp-result (make-instance relation))
      (send temp-result set-members
            shared-result-members)
      (if (symbol? result)
       (begin
        (send temp-result set-self result)
        temp-result)
       (begin
        (send temp-result set-self 'temp-result)
        (send temp-result display-yourself))
      ))
     (#T
      (while (not (null? shared-result-members))
       (fluid-let ((tvar (send (eval rooms) get-var)))
        (eval '(set! ,(fluid tvar)
                 (make-instance ,record-type)))
        (send (eval (fluid tvar)) set-self (fluid tvar))
        (send (eval (fluid tvar)) copy-from
              (eval '(eval
                ,(car shared-result-members))))
        (set! result-members (append
          result-members (list (fluid tvar))))
        (set! shared-result-members
          (cdr shared-result-members))
      ))
      (send result set-members result-members)
      (send result display-yourself))))
   (#T
    (list 'cannot 'union 'unlike 'relations))
  )
))



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; COMPILE-CLASS RELATION
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(compile-class relation)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; RECORD
; - Description
;   - intended to be used as an ancestor to every
;     user-defined record type
;   - note that records may consist only of subobjects
; - Superclasses
;   - OBJECT, GENERIC
; - Class Variables
;   - none
; - Instance Variables
;   - none
; - Methods
;   - (send-if-handles x IS-RECORD?)
;     - #T if x is of type record, #F otherwise
;   - (send x MEET-SELECT-CRITERIA? which op what)
;     - used by the SELECT method of RELATION
;     - which indicates which subobject
;     - op indicates which operation
;     - what indicates what to compare the subobject to
;     - for example, (send x MEET-SELECT-CRITERIA?
;                             subobject1 is-equal-to? y)
;       - #T if subobject1 of x is-equal-to? y,
;         otherwise #F
;   - (send x COPIES-FROM other1 other2)
;     - similar to the COPY-FROM inherited from GENERIC
;     - used by CARTESIAN-PRODUCT
;     - makes x a copy of the fields of other1 & other2
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-class record
  (mixins object generic)
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; RECORD IS-RECORD?
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (record is-record?) () #T)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; RECORD MEET-SELECT-CRITERIA?
; - WHICH is which subobject in the record
; - OP is which operation (such as EQUAL-TO?)
; - WHAT is what value (object) to compare the WHICH to
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (record meet-select-criteria?)
               (which op what)
  (if (eval '(send ,(eval which) ,op ,what))
    #T
    #F)
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; RECORD COPIES-FROM
; - similar to the inherited COPY-FROM, except that this
;   copies a record from two other records
; - used by CARTESIAN-PRODUCT
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (record copies-from) (rec1 rec2)
 (let ((my-instvar-list (cdr instvar-list))
       (rec1-instvar-list
         (cdr (send rec1 get-instvar-list)))
       (rec2-instvar-list
         (cdr (send rec2 get-instvar-list)))
       (next-var nil))
  (while (not (null? rec1-instvar-list))
    (set! next-var (car my-instvar-list))
    (cond
     ((equal? (cadr next-var) 'subobject)
      (fluid-let ((tvar (send (eval rooms) get-var)))
       (eval '(set! ,(car next-var) (fluid tvar)))
       (eval '(set! ,(fluid tvar)
                (make-instance ,(caddr next-var))))
       (send (eval (fluid tvar)) set-self (fluid tvar))
       (send (eval (fluid tvar)) copy-from
             (eval (eval
               '(send ,rec1 ,(concat 'get-
                                     (car next-var))))))))
     (#T
      (eval '(set! ,(car next-var) nil)))
    )
    (set! rec1-instvar-list (cdr rec1-instvar-list))
    (set! my-instvar-list (cdr my-instvar-list)))
  (while (not (null? rec2-instvar-list))
    (set! next-var (car my-instvar-list))
    (cond
```

```
      ((equal? (cadr next-var) 'subobject)
       (fluid-let ((tvar (send (eval rooms) get-var)))
        (eval '(set! ,(car next-var) (fluid tvar)))
        (eval '(set! ,(fluid tvar)
                 (make-instance ,(caddr next-var))))
        (send (eval (fluid tvar)) set-self (fluid tvar))
        (send (eval (fluid tvar)) copy-from (eval (eval
              '(send ,rec2 ,(concat
                 'get- (car next-var)))))))))
      (#T
       (eval '(set! ,(car next-var) nil)))
     )
     (set! rec2-instvar-list (cdr rec2-instvar-list))
     (set! my-instvar-list (cdr my-instvar-list)))
))



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; COMPILE-CLASS RECORD
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(compile-class record)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; MACROS & FUNCTIONS TO CREATE OBJECTS & "NULL" OBJECTS
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(macro create-database (lambda (e)
  '(make-instance database 'self ',(cadr e))))

(macro create-relation (lambda (e)
  '(make-instance relation 'self ',(cadr e))))

(macro create-record (lambda (e)
  '(make-instance record 'self ',(cadr e))))

(define null-database
  (create-database     null-database))

(define null-relation
  (create-relation     null-relation))

(define null-record
  (create-record       null-record))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ROOMS-10.PCS - section 10 of ROOMS in PC Scheme
;  This section contains the descriptions for
;  - database user's records
;    - EMP-REC
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; EMP-REC
; - Description
;   - Employee Record's
; - Superclasses
;   - RECORD
; - Class Variables
;   - INSTVAR-LIST, DISPLAY-LIST, EQUALITY-LIST
; - Instance Variables
;   - EMP-NAME, subobject of type person-name
;   - EMP-ADDR, subobject of type addr
;   - EMP-WIDGET, subobject of type widget
;   - EMP-SALARY, subobject of type salary
; - Methods
;   - (send-if-handles x IS-EMP-REC?)
;     - #T if x is of type emp-rec, #F otherwise
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-class emp-rec
  (mixins record)
  (classvars
    (instvar-list  '((self instvar aatom)
                     (emp-name subobject person-name)
                     (emp-addr subobject addr)
                     (emp-widget subobject widget)
                     (emp-salary subobject salary)))
    (display-list  '((emp-name subobject)
                     (emp-addr subobject)
                     (emp-widget subobject)
                     (emp-salary subobject)))
    (equality-list '((emp-name subobject)
                     (emp-salary subobject)
                     (emp-addr subobject)
                     (emp-widget subobject))))
  (instvars  (emp-name 'null-person-name)
             (emp-addr 'null-addr)
             (emp-widget 'null-widget)
             (emp-salary 'null-salary))
  (options gettable-variables settable-variables
           inittable-variables)
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; EMP-REC IS-EMP-REC?
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (emp-rec is-emp-rec?) () #T)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; COMPILE-CLASS EMP-REC
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(compile-class emp-rec)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; MACROS & FUNCTIONS TO CREATE OBJECTS & "NULL" OBJECTS
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(macro create-emp-rec (lambda (e)
  '(make-instance emp-rec 'self ',(cadr e))))

(define null-emp-rec
  (create-emp-rec      null-emp-rec))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ROOMS-20.PCS - section 20 of ROOMS in PC Scheme
; - this section contains the descriptions for
;   all user-defined fields
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; NAME
; - Description
;   - contains a single value (atom, string, list, etc)
;     which is the "name" of something
; - Superclasses
;   - OBJECT, GENERIC
; - Class Variables
;   - INSTVAR-LIST
;   - DISPLAY-LIST
;   - EQUALITY-LIST
; - Instance Variables
;   - VALUE
; - Methods
;   - (send-if-handles x IS-NAME?)
;     - #T if x is of type name, #F otherwise
;   - (send x CHANGE-TO new-name)
;     - change value to new-name
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-class name
  (mixins object generic)
  (classvars  (instvar-list  '((self instvar aatom)
                               (value instvar vvar)))
              (display-list  '((value instvar)))
              (equality-list '((value instvar))))
  (instvars   (value nil))
  (options gettable-variables settable-variables
           inittable-variables)
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; NAME IS-NAME?
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (name is-name?) () #T)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; NAME CHANGE-TO
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (name change-to) (new-name)
  (set! value new-name)
  new-name
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; COMPILE-CLASS NAME
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(compile-class name)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; LAST-NAME
; - Description
;   - contains a last name
; - Superclasses
;   - NAME
; - Class Variables
;   - none
; - Instance Variables
;   - none
; - Methods
;   - (send-if-handles x IS-LAST-NAME?)
;     - #T if x is of type last-name, #F otherwise
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-class last-name
  (mixins name)
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; LAST-NAME IS-LAST-NAME?
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (last-name is-last-name?) () #T)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; COMPILE-CLASS LAST-NAME
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(compile-class last-name)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; FIRST-NAME
; - Description
;   - contains a first-name
; - Superclasses
;   - NAME
; - Class Variables
;   - none
; - Instance Variables
;   - none
; - Methods
;   - (send-if-handles x IS-FIRST-NAME?)
;     - #T if x is of type first-name, #F otherwise
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-class first-name
  (mixins name)
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; FIRST-NAME IS-FIRST-NAME?
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (first-name is-first-name?) () #T)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; COMPILE-CLASS FIRST-NAME
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(compile-class first-name)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; MIDDLE-INITIAL
; - Description
;   - contains a middle-initial
; - Superclasses
;   - NAME
; - Class Variables
;   - none
; - Instance Variables
;   - none
; - Methods
;   - (send-if-handles x IS-MIDDLE-INITIAL?)
;     - #T if x is of type middle-initial, #F otherwise
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-class middle-initial
  (mixins name)
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; MIDDLE-INITIAL IS-MIDDLE-INITIAL?
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (middle-initial is-middle-initial?) ()
  #T)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; COMPILE-CLASS MIDDLE-INITIAL
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(compile-class middle-initial)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; PERSON-NAME
; - Description
;   - contains a persons name
;     (last, first, middle initial)
; - Superclasses
;   - OBJECT, GENERIC
; - Class Variables
;   - INSTVAR-LIST, DISPLAY-LIST, EQUALITY-LIST
; - Instance Variables
;   - PN-LAST-NAME, a subobject of type last-name
;   - PN-FIRST-NAME, a subobject of type first-name
;   - PN-MIDDLE-INITIAL, a subobject of type
;     middle-initial
; - Methods
;   - (send-if-handles x IS-PERSON-NAME?)
;     - #T if x is of type person-name, #F otherwise
;   - (send x CHANGE-LAST-NAME-TO new-last-name)
;     - calls (send last-name change-to new-last-name)
;   - (send x CHANGE-FIRST-NAME-TO new-first-name)
;     - calls (send first-name change-to new-first-name)
;   - (send x CHANGE-MIDDLE-INITIAL-TO new-mi)
;     - calls (send middle-initial change-to new-mi)
;   - (send x CHANGE-TO new-name)
;     - calls
;       (send self change-last-name-to (car new-name))
;       (send self change-first-name-to (cadr new-name))
;       (send self change-middle-initial-to
;             (caddr new-name))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-class person-name
  (mixins object generic)
  (classvars
    (instvar-list
      '((self instvar aatom)
        (pn-last-name subobject last-name)
        (pn-first-name subobject first-name)
        (pn-middle-initial subobject middle-initial)))
    (display-list '((pn-last-name subobject)
                    (pn-first-name subobject)
                    (pn-middle-initial subobject)))
    (equality-list '((pn-last-name subobject)
                     (pn-first-name subobject)
                     (pn-middle-initial subobject))))
  (instvars  (pn-last-name 'null-last-name)
             (pn-first-name 'null-first-name)
             (pn-middle-initial 'null-middle-initial))
  (options gettable-variables settable-variables
           inittable-variables)
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; PERSON-NAME IS-PERSON-NAME?
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (person-name is-person-name?) () #T)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; PERSON-NAME CHANGE-LAST-NAME-TO
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (person-name change-last-name-to)
               (new-last-name)
  (if (not (null? pn-last-name))
    (send (eval pn-last-name) change-to new-last-name)
    #F)
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; PERSON-NAME CHANGE-FIRST-NAME-TO
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (person-name change-first-name-to)
               (new-first-name)
  (if (not (null? pn-first-name))
    (send (eval pn-first-name) change-to new-first-name)
    #F)
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; PERSON-NAME CHANGE-MIDDLE-INITIAL-TO
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (person-name change-middle-initial-to)
               (new-mi)
  (if (not (null? pn-middle-initial))
    (send (eval pn-middle-initial) change-to new-mi)
    #F)
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; PERSON-NAME CHANGE-TO
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (person-name change-to) (new-name)
  (if (and (equal? '3 (length new-name))
           (atom? (car new-name))
           (atom? (cadr new-name))
           (atom? (caddr new-name)))
    (append
      (list (send (eval self) change-last-name-to
                  (car new-name)))
      (list (send (eval self) change-first-name-to
                  (cadr new-name)))
      (list (send (eval self) change-middle-initial-to
                  (caddr new-name))))
    nil)
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; COMPILE-CLASS PERSON-NAME
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(compile-class person-name)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ADDR
; - Description
;   - contains an address
;     (street, city, state, zip)
; - Superclasses
;   - OBJECT, GENERIC
; - Class Variables
;   - INSTVAR-LIST, DISPLAY-LIST, EQUALITY-LIST
; - Instance Variables
;   - STREET, a subobject of type name
;   - CITY, a subobject of type name
;   - STATE, a subobject of type name
;   - ZIP, a subobject of type name
; - Methods
;   - (send-if-handles x IS-ADDR?)
;     - #T if x is of type addr, #F otherwise
;   - (send x CHANGE-STREET-TO new-street)
;     - calls (send street change-to new-street)
;   - (send x CHANGE-CITY-TO new-city)
;     - calls (send city change-to new-city)
;   - (send x CHANGE-STATE-TO new-state)
;     - calls (send state change-to new-state)
;   - (send x CHANGE-ZIP-TO new-zip)
;     - calls (send zip change-to new-zip)
;   - (send x CHANGE-TO new-addr)
;     - calls
;       (send self change-street-to (car new-addr))
;       (send self change-city-to (cadr new-addr))
;       (send self change-state-to (caddr new-addr))
;       (send self change-zip-to (cadddr new-addr))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-class addr
  (mixins object generic)
  (classvars
    (instvar-list
     '((self instvar aatom)  (street subobject name)
       (city subobject name) (state subobject name)
       (zip subobject name)))
    (display-list
     '((street subobject) (city subobject)
       (state subobject)  (zip subobject)))
    (equality-list
     '((zip subobject)  (state subobject)
       (city subobject) (street subobject))))
  (instvars (street 'null-street) (city 'null-city)
            (state 'null-state)   (zip 'null-zip))
  (options gettable-variables settable-variables
           inittable-variables)
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ADDR IS-ADDR?
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (addr is-addr?) () #T)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ADDR CHANGE-STREET-TO
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (addr change-street-to) (new-street)
  (if (not (null? street))
    (send (eval street) change-to new-street)
    #F)
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ADDR CHANGE-CITY-TO
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (addr change-city-to) (new-city)
  (if (not (null? city))
    (send (eval city) change-to new-city)
    #F)
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ADDR CHANGE-STATE-TO
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (addr change-state-to) (new-state)
  (if (not (null? state))
    (send (eval state) change-to new-state)
    #F)
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ADDR CHANGE-ZIP-TO
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (addr change-zip-to) (new-zip)
  (if (not (null? zip))
    (send (eval zip) change-to new-zip)
    #F)
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ADDR CHANGE-TO
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (addr change-to) (new-addr)
  (if (and (equal? '4 (length new-addr))
           (atom? (car new-addr))
           (atom? (cadr new-addr))
           (atom? (caddr new-addr))
           (atom? (cadddr new-addr)))
    (append
      (list (send (eval self) change-street-to
                  (car new-addr)))
      (list (send (eval self) change-city-to
                  (cadr new-addr)))
      (list (send (eval self) change-state-to
                  (caddr new-addr)))
      (list (send (eval self) change-zip-to
                  (cadddr new-addr))))
    nil)
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ADDR DISPLAY-YOURSELF
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (addr display-yourself) ()
  (append (send (eval street) display-yourself)
          (send (eval city)   display-yourself)
          (send (eval state)  display-yourself)
          (send (eval zip)    display-yourself))
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; COMPILE-CLASS ADDR
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(compile-class addr)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; WIDGET
; - Description
;   - looks amazingly like a telephone number,
;     - x-coord == area code
;     - y-coord == exchange
;     - z-coord == extension
;   - note: does not use generic routines (by choice...)
; - Superclasses
;   - OBJECT
; - Class Variables
;   - none
; - Instance Variables
;   - X-COORD, Y-COORD, & Z-COORD
; - Methods
;   - (send-if-handles x IS-WIDGET?)
;     - #T if x is of type widget, #F otherwise
;   - (send x COPY-FROM other)
;     - if (other is of type widget)
;       set instvars of x to the instvars of other
;   - (send x DISPLAY-YOURSELF)
;     - displays x-coord, y-coord, & z-coord
;   - (send x EQUAL-TO? y)
;     - #T if (y is of type widget) and (the instance
;       variables of x are equal to the instance
;       variables of y), otherwise #F
;   - (send x SET-WIDGET coords)
;     - sets x-coord to (car coords),
;       y-coord to (cadr coords), &
;       z-coord to (caddr coords)
;   - (send x CHANGE-TO new-widget)
;     - calls (send self set-widget new-widget)
;   - (send x CHANGE-X-TO new-x)
;     - sets x-coord to new-x
;   - (send x CHANGE-Y-TO new-y)
;     - sets y-coord to new-y
;   - (send x CHANGE-Z-TO new-z)
;     - sets z-coord to new-z
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-class widget
  (mixins object)
  (instvars  (x-coord nil) (y-coord nil) (z-coord nil))
  (options gettable-variables settable-variables
           inittable-variables)
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; WIDGET IS-WIDGET?
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (widget is-widget?) () #T)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; WIDGET COPY-FROM
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (widget copy-from) (other)
  (cond
   ((send-if-handles other is-widget?)
    (set! x-coord (send other get-x-coord))
    (set! y-coord (send other get-y-coord))
    (set! z-coord (send other get-z-coord))
    (eval self))
   (#T
    nil))
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; WIDGET DISPLAY-YOURSELF
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (widget display-yourself) ()
  (append (list x-coord) (list y-coord) (list z-coord))
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; WIDGET EQUAL-TO?
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (widget equal-to?) (other)
  (if (and (send-if-handles other is-widget?)
           (equal? x-coord (send other get-x-coord))
           (equal? y-coord (send other get-y-coord))
           (equal? z-coord (send other get-z-coord)))
    #T
    #F)
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; WIDGET SET-WIDGET
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (widget set-widget) (coords)
  (if (and (equal? (length coords) '3)
           (atom? (car coords))
           (atom? (cadr coords))
           (atom? (caddr coords)))
    (append (list (set! x-coord (car coords)))
            (list (set! y-coord (cadr coords)))
            (list (set! z-coord (caddr coords))))
    nil)
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; WIDGET CHANGE-TO
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (widget change-to) (coords)
  (send (eval self) set-widget coords)
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; WIDGET CHANGE-X-TO
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (widget change-x-to) (new-x)
  (set! x-coord new-x)
  new-x
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; WIDGET CHANGE-Y-TO
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (widget change-y-to) (new-y)
  (set! y-coord new-y)
  new-y
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; WIDGET CHANGE-Z-TO
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (widget change-z-to) (new-z)
  (set! z-coord new-z)
  new-z
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; COMPILE-CLASS WIDGET
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(compile-class widget)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; SALARY
; - Description
;   - integer field
;   - note: does not use generic routines (by choice...)
; - Superclasses
;   - OBJECT
; - Class Variables
;   - none
; - Instance Variables
;   - VALUE
; - Methods
;   - (send-if-handles x IS-SALARY?)
;     - #T if x is of type salary, #F otherwise
;   - (send x COPY-FROM other)
;     - if (other is of type salary)
;       set value of x to the value of other
;   - (send x DISPLAY-YOURSELF)
;     - displays value
;   - (send x EQUAL-TO? y)
;     - if (y is of type salary) and
;       (the instance value of x is equal to the value
;       of y) #T, otherwise #F
;   - (send x SET-SALARY new-value)
;     - sets value to new-value
;   - (send x CHANGE-TO new-value)
;     - calls (send self set-salary new-value)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-class salary
  (mixins object)
  (instvars  (value 0))
  (options gettable-variables settable-variables
           inittable-variables)
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; SALARY IS-SALARY?
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (salary is-salary?) () #T)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; SALARY COPY-FROM
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (salary copy-from) (other)
  (cond
   ((send-if-handles other is-salary?)
    (set! value (send other get-value))
    (eval self))
   (#T
    nil))
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; SALARY DISPLAY-YOURSELF
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (salary display-yourself) ()
  (list value)
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; SALARY EQUAL-TO?
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (salary equal-to?) (other)
  (if (and (send-if-handles other is-salary?)
           (equal? value (send other get-value)))
    #T
    #F)
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; SALARY SET-SALARY
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (salary set-salary) (new-value)
  (if (not (number? new-value))
    nil
    (set! value new-value))
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; SALARY CHANGE-TO
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (salary change-to) (new-value)
  (send (eval self) set-salary new-value)
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; COMPILE-CLASS SALARY
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(compile-class salary)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; MACROS & FUNCTIONS TO CREATE OBJECTS & "NULL" OBJECTS
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(macro create-name (lambda (e)
  '(make-instance name 'self ',(cadr e))))

(macro create-last-name (lambda (e)
  '(make-instance last-name 'self ',(cadr e))))

(macro create-first-name (lambda (e)
  '(make-instance first-name 'self ',(cadr e))))

(macro create-middle-initial (lambda (e)
  '(make-instance middle-initial 'self ',(cadr e))))

(macro create-person-name (lambda (e)
  '(make-instance person-name 'self ',(cadr e))))

(macro create-addr (lambda (e)
  '(make-instance addr 'self ',(cadr e))))

(macro create-widget (lambda (e)
  '(make-instance widget 'self ',(cadr e))))

(macro create-salary (lambda (e)
  '(make-instance salary 'self ',(cadr e))))

(define null-name (create-name null-name))

(define null-last-name
  (create-last-name null-last-name))

(define null-first-name
  (create-first-name null-first-name))

(define null-middle-initial
  (create-middle-initial null-middle-initial))

(define null-person-name
  (create-person-name null-person-name))

(define null-addr (create-addr null-addr))

(define null-widget (create-widget null-widget))

(define null-salary (create-salary null-salary))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; UTILS.PCS - utilities used by ROOMS
; - this section contains the descriptions for
;   - (concat x y)
;     - concatenates x & y together
;   - (embedded-member? x y)
;     - determines if x is contained anywhere within y
;   - (list? x)
;     - determines if x is a list
;   - (pprint x)
;     - prints each member of x on a separate line
;   - (subset? x y)
;     - determines if x is a subset of y
;   - (while condition statements)
;     - while condition is true, executes statements
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; (CONCAT x y)
; - concatenates x and y together,
;   - x and y must be atoms
; - ex, (concat 'a 'b)    = 'ab
;       (concat 'ab 'xyz) = 'abxyz
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define concat (lambda (x y)
  (implode (append (explode x) (explode y)))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; (EMBEDDED-MEMBER? x y)
; - determines if x appears anywhere in y
;   - x can be an atom or a list, y must be a list
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (embedded-member? x y)
 (let ((return-value #F)
       (finished #F))
  (if (list? y)
    (while (not finished)
      (if (or (member x y)
              (and (list? (car y))
                   (embedded-member? x (car y))))
        (begin
         (set! return-value #T)
         (set! finished #T))
        (begin
          (set! y (cdr y))
          (if (null? y) (set! finished #T))))
    ))
  return-value
))



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; (LIST? x)
; - determines if x is a list
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(macro list? (lambda (e) '(not (atom? ,(cadr e)))))



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; (PPRINT x)
; - display each member of the list x on a separate line
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (pprint x)
  (while (>? (length x) 1)
    (print (car x))
    (set! x (cdr x))
  )
  (newline)
  (car x)
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; (SUBSET? l1 l2)
; - #T if l1 is a subset of l2
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (subset? l1 l2)
 (cond
  ((null? l1)                #T)
  ((member (car l1) l2)      (subset? (cdr l1) l2))
  (t                         #F)
))



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; (WHILE condition statements)
; - while condition is #T, execute statements
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(macro while
  (lambda (e)
    '(do #!null ((not,(cadr e))),@(cddr e))
))
```

APPENDIX C

THE SOURCE CODE FOR THE IMPLEMENTATION OF ROOMS IN EOOPS

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ROOMS-00.EOO - section 00 of ROOMS
; - this section contains the descriptions for
;   - GENERIC METHODS
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; GENERIC METHODS
; - Description
;   - Various "default" methods which can be
;     instantiated for other classes
; - Methods
;   - (send x COPY-FROM other)
;     - makes x a "copy" of other by setting all
;       instance variables of x which are also instance
;       variables of other to the same value, and calls
;       (send x SUPERCLASS.COPY-FROM other) for every
;       superclass of x's class which is also a
;       superclass of y's class
;   - (send x DISPLAY-YOURSELF)
;     - Concatenates the display value of every class
;       variable, instance variable, and the result of
;       (send x SUPERCLASS.DISPLAY-YOURSELF) for every
;       superclass
;   - (send x EQUAL-TO? other)
;     - if x & other are of the same class,
;       - then compare the values of x's and other's
;         instance variables, and the result of
;         (send x SUPERCLASS.EQUAL-TO? other) for every
;         superclass to determine equality
;       - otherwise false
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; GENERIC COPY-FROM
; - if self & other are not of the same class
;   - then for every class variable of self that is also
;     a class variable of other, set self's class
;     variable to that of other's
; - for every instance variable of self that is also an
;   instance variable of other, set self's instance
;   variable to that of other's
; - for every superclass of self that is also a
;   superclass of other, send self the message
;   superclass.copy-from other
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-generic-method copy-from (other) ()
 (let ((my-class (send self class-of-object?))
       (my-class-variables
         (send my-class get-class-variables))
       (my-instance-variables
         (send my-class get-instance-variables))
       (my-superclasses
         (send my-class get-superclasses))
       (other-class (send other class-of-object?))
       (other-class-variables
         (send other-class get-class-variables))
       (other-instance-variables
         (send other-class get-instance-variables)))
  (if (not (equal? my-class other-class))
    (while (not (null? my-class-variables))
      (if (embedded-member? (caar my-class-variables)
                            other-class-variables)
        (set! (eval (caar my-class-variables))
          (send other (concat
            '(get-) (caar my-class-variables)))))
      (set! my-class-variables
        (cdr my-class-variables))))
  (while (not (null? my-instance-variables))
    (if (embedded-member? (caar my-instance-variables)
                          other-instance-variables)
      (set! (eval (caar my-instance-variables))
        (send other (concat
          '(get-) (caar my-instance-variables)))))
    (set! my-instance-variables
      (cdr my-instance-variables)))
  (while (not (null? my-superclasses))
    (if (embedded-member? (car my-superclasses)
                          other-superclasses)
      (send self (concat (car my-superclasses)
                         '(.copy-from))
        other)))
))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; GENERIC DISPLAY-YOURSELF
; - this version puts parens around subobjects
; - essentially,
;   - for x in (send self get-class-variables),
;     display x
;   - for x in (send self get-instance-variables),
;     display x
;   - for x in (send self get-superclasses),
;     (send self superclass.display-yourself)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-generic-method display-yourself () ()
 (let ((return-message nil)
       (my-class (send self class-of-object?))
       (my-class-variables
         (send my-class get-class-variables))
       (my-instance-variables
         (send my-class get-instance-variables))
       (my-superclasses
         (send my-class get-superclasses)))
  (while (not (null? my-class-variables))
    (set! return-message
      (append return-message
              (list (eval (caar my-class-variables)))))
    (set! my-class-variables (cdr my-class-variables)))
  (while (not (null? my-instance-variables))
    (set! return-message
      (append return-message
              (list (eval
                      (caar my-instance-variables)))))
    (set! my-instance-variables
      (cdr my-instance-variables)))
  (while (not (null? my-superclasses))
    (set! return-message
      (append return-message
              (list (send self
                      (concat (car my-superclasses)
                              '(.display-yourself))))))
    (set! my-superclasses (cdr my-superclasses)))
  return-message
))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; GENERIC METHODS EQUAL-TO?
; - self & other must be of the same class
; - assume the objects are equal (i.e., set return-value
;   to TRUE)
; - essentially,
;   - for x in (send self get-instance-variables),
;     compare self's & other's value for x
;   - for x in (send self get-superclasses),
;     (send self superclass.equal-to? other)
;   discontinue if any pair of values are not equal
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-generic-method equal-to? (other) ()
 (let ((return-value #T)
       (my-class (send self class-of-object?))
       (my-instance-variables
         (send my-class get-instance-variables))
       (my-superclasses
         (send my-class get-superclasses)))
  (while (and (not (null? my-instance-variables))
              (not (null? return-value)))
    (if (not (equal?
               (eval (caar my-instance-variables))
               (send other (concat
                 '(get-)
                 (caar my-instance-variables)))))
      (set! return-value #F))
    (set! my-class-variables (cdr my-class-variables)))
  (while (and (not (null? my-superclasses))
              (not (null? return-value)))
    (if (not (send self (concat (car my-superclasses)
                                '(.equal-to?))))
    (set! return-value #F)
    (set! my-superclasses (cdr my-superclasses)))
  return-value
))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ROOMS-01.EOO - section 01 of ROOMS in EOOPS
;  - this section contains the descriptions for
;   - all database "overhead" classes
;     - DATABASE, RELATION, & RECORD
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; DATABASE
; - Description
;   - consists of a collection of relations
; - Superclasses
;   - none
; - Class Variables
;   - none
; - Instance Variables
;   - MEMBERS
;     - a list of all the relations that are in the
;       database; ex, '(R S T)
; - Methods
;   - (send x DISPLAY-YOURSELF)
;     - displays the names of the relations that are in
;       the database
;   - (send x ADD-MEMBFR member-to-add)
;     - adds member-to-add to the member list if it is
;       of type relation
;   - (send x REMOVE-MEMBER member-to-remove)
;     - removes member-to-remove from the member list
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-class database
  (instance-variables  (members nil)
   options gettable settable)
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; DATABASE DISPLAY-YOURSELF
; - display a list of all the relations in the database
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(database define-method display-yourself ()
  members
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; DATABASE ADD-MEMBER
; - if member-to-add is of type relation and it is not
;   already in the members list, then add it to the
;   members list
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(database define-method add-member (member-to-add)
  (if eq? (send member-to-add class-of-object?)
          'relation)
    (if (not (member member-to-add members))
      (set! members (append members
                            (list member-to-add)))
      members)
    nil)
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; DATABASE REMOVE-MEMBER
; - delete member-to-remove from members list
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(database define-method remove-member (member-to-remove)
  (set! members (delete! member-to-remove members))
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; RELATION
; - Description
;   - Intended to be used as an ancestor to every
;     user-defined relation type
;   - Relations consist of a collection of records
; - Superclasses
;   - none
; - Class Variables
;   - none
; - Instance Variables
;   - MEMBERS
;     - a list of all the records that are in this
;       relation; ex, '(e1 e2 e4 e7)
;   - RECORD-TYPE
;     - the class name of records that can be in this
;       relation; ex, 'emp-rec
; - Methods
;   - (send x DISPLAY-YOURSELF)
;     - displays each record that is a member of the
;       relation
;   - (send x ADD-RECORD record-to-add)
;     - add the record record-to-add to the relation
;       (i.e., to the member list)
;     - record-to-add must be a symbol
;       - i.e., if e1 is the record to be added, must
;         use 'e1 or x where (eval x) = 'e1, e1 will NOT
;         work
;     - record-to-add must be of type RECORD-TYPE
;   - (send x GET-RECORDS)
;     - returns members, the list of records
;   - (send x REMOVE-RECORD record-to-remove)
;     - removes record-to-remove from the relation
;       (i.e., from the member list)
;   - relational algebra operations
;     - for all operations
;       - if result is nil
;         - then build & display a temporary relation
;       - if result is a symbol
;         - then build & return a temporary relation
;       - if result is a relation
;         - then build & display it
;     - (send x CARTESIAN-PRODUCT other result
;                                   new-rec-class)
;       - make result the cartesian-product of self &
;         other
;       - if new-rec-class is nil
;         - then a gensym variable name is used as the
;           name of the new record type
;       - if new-rec-class is a symbol
;         - then that symbol is used as the name of the
```

```
;            new record type
;          - if new-rec-class is already a type of record
;            - then it is used as the record-type of the
;              result relation
;      - (send x DIFFERENCE other result)
;        - make result the difference of self & other
;      - (send x FAST-INTERSECT other result)
;        - make result the intersection of self & other,
;          using the EQUAL-TO? method to determine which
;          records to pick
;      - (send x INTERSECT other result)
;        - make result the intersection of self & other,
;          using the "classic" formula
;          result = (self - (self - other)),
;          implemented as
;          - temp-relation = (self - other)
;          - result = (self - temp-relation)
;      - (send x PROJECT (field-list) result
;                           new-rec-class)
;        - make result the projection of field-lists from
;          self, where new-rec-class is the name of the
;          record class of the result records
;        - if new-rec-class is nil
;          - then a gensym variable name is used as the
;            name of the new record type
;        - if new-rec-class is a symbol
;          - then that symbol is used as the name of the
;            new record type
;        - if new-rec-class is already a type of record
;          - then it is used as the record-type of the
;            result relation
;      - (send x SELECT which op what result)
;        - make result those records of self that meet
;          the selection criteria of which, op, & what
;      - (send x UNION other result)
;        - make result the union of self & other
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-class relation
  (instance-variables  (members nil)  (record-type nil)
   options gettable settable)
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; RELATION DISPLAY-YOURSELF
; - concatenate the results of
;   (send record DISPLAY-YOURSELF)
;   for each record in the relation
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(relation define-method display-yourself ()
 (let ((member-list members)
       (return-message nil))
  (while (not (null? member-list))
    (set! return-message
      (append return-message
              (list (send (eval (car member-list))
                                  display-yourself))))
    (set! member-list (cdr member-list))
  )
  (pprint return-message)
))



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; RELATION ADD-RECORD
; - if record-to-add is of the appropriate record type
;   and is not already a member of the relation, then
;   add it to the members list
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(relation define-method add-record (record-to-add)
  (if (and (symbol? record-to-add)
           (equal? (send record-to-add class-of-object?)
                   record-type))
    (if (not (member record-to-add members))
      (set! members
        (append members
                (eval '(list (quote ,record-to-add)))))
      members)
    nil)
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; RELATION GET-RECORDS
; - display the members of the relation
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(relation define-method get-records ()
  members
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; RELATION REMOVE-RECORD
; - remove record-to-remove from the members list
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(relation define-method remove-record (record-to-remove)
  (set! members (delete! record-to-remove members))
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; RELATION CARTESIAN-PRODUCT
; - build the new record class if necessary
; - instantiate the result-relation if necessary
; - set the record-type of the result-relation to the
;   new record class if necessary
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(relation define-method cartesian-product
          (other result new-rec-class)
 (let ((result-relation nil)
       (temp-rec nil)
       (my-members nil)
       (others-members nil))
  (if (or (null? new-rec-class) (symbol? new-rec-class))
   (begin
    (if (null? new-rec-class)
        (set! new-rec-class (gensym)))
    (let ((my-record-type (send self get-record-type))
          (other-record-type
            (send other get-record-type))
          (new-class-variables nil)
          (new-instance-variables nil)
          (new-superclasses nil))
     (set! new-class-variables (append
       (send my-record-type get-class-variables)
       (send other-record-type get-class-variables)))
     (set! new-instance-variables (append
       (send my-record-type get-instance-variables)
       (send other-record-type get-instance-variables)))
     (set! new-superclasses (append
       (send my-record-type get-superclasses)
       (send other-record-type get-superclasses)))
     (define-class new-rec-class
       (append '(class-variables) new-class-variables)
       (append '(instance-variables)
               new-instance-variables)
       (append '(superclasses) new-superclasses)
  ))
; new-rec-class has now been created
  (cond
   ((null? result)
    (set! result-relation (gensym))
    (send relation make-instance result-relation))
   ((symbol? result)
    (send relation make-instance result-relation))
   (#T
    (set! result-relation result)))
  (if (not (equal? (send result-relation
                         get-record-type)
                   new-rec-class))
```

```
    (send result-relation set-record-type
       new-rec-class))
; result-relation has now been created,
;  with its record-type set to the new-rec-class
  (set! others-members (send other get-members))
  (while (not (null? others-members))
   (set! my-members members)
   (while (not (null? my-members))
    (set! temp-rec (gensym))
    (send new-rec-class make-instance (eval temp-rec))
    (send (eval temp-rec) copy-from (car my-members)
                          (car others-members))
    (send result-relation add-member temp-rec)
    (set! my-members (cdr my-members)))
   (set others-members (cdr others-members)))
; result-relation is now the result of the cartesian
; product
  (cond
   ((null? result)
    (send result-relation display-yourself))
   ((symbol? result)
    result-relation)
   (#T
    (set! result result-relation)
    (send result display-yourself)
))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; RELATION DIFFERENCE
; - choose records that are in self but not in other,
;   using EQUAL-TO? to determine equality
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(relation define-method difference (other result)
 (let ((my-members members)          (other-members nil)
       (shared-result-members nil)  (result-members nil)
       (add-flag #T)                (temp-result nil))
  (cond
   ((and (equal? (send other class-of-object?)
                 'relation)
         (or (null? result)
             (symbol? result)
             (equal? (send result class-of-object?)
                     'relation)))
    (set! other-members (send other get-members))
    (while (not (null? my-members))
      (while (not (null? other-members))
        (cond
         ((send (eval (car my-members)) equal-to?
                (eval (car other-members)))
          (set! other-members nil)
          (set! add-flag nil))
         (#T
          (set! other-members (cdr other-members)))
      ))
      (if add-flag
        (set! shared-result-members
          (append shared-result-members
                  (list (car my-members))))
        (set! add-flag #T))
      (set! my-members (cdr my-members))
      (set! other-members (send other get-members))
    )
    (send relation make-instance temp-result)
    (send temp-result set-members shared-result-members)
    (cond
     ((null? result)
       (send temp-result display-yourself)
      (symbol? result)
       temp-result)
      (#T
       (send result copy-from temp-result)
       (send result display-yourself))))
   (#T
    (list 'cannot 'difference 'unlike 'relations))
  )
))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; RELATION INTERSECT
; - use the "classic" formula for intersection
;   (self - (self - other))
; - no need to worry about what result is as the
;   difference method will take care of that
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(relation define-method intersect (other result)
 (let ((temp-relation nil))
  (cond
   ((equal? (send other class-of-object?) 'relation)
    (send relation make-instance temp-relation)
    (self difference other temp-relation)
    (self difference temp-relation result)
   (#T
    (list 'cannot 'intersect 'unlike 'relations))
  )
))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; RELATION FAST-INTERSECT
; - choose every record that is a member of both self &
;   other, using EQUAL-TO? to determine equality
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(relation define-method fast-intersect (other result)
 (let ((my-members members)
       (other-members (send other get-members))
       (shared-result-members nil)
       (result-members nil)
       (add-flag #F)
       (temp-result nil))
  (cond
   ((and (equal? (send other class-of-object?)
                 'relation)
         (or (null? result)
             (symbol? result)
             (equal? (send result class-of-object?)
                     'relation)))
    (while (not (null? other-members))
      (while (not (null? my-members))
        (cond
         ((send (eval (car my-members)) equal-to?
                (eval (car other-members)))
          (set! my-members nil)
          (set! add-flag #T))
         (#T
          (set! my-members (cdr my-members)))))
      (cond
       (add-flag
        (set! shared-result-members
          (append shared-result-members
                  (list (car other-members))))
        (set! add-flag #F)))
      (set! my-members members)
      (set! other-members (cdr other-members)))
    (send relation make-instance temp-result)
    (send temp-result set-members shared-result-members)
    (cond
     ((null? result)
      (send temp-result display-yourself))
     (symbol? result)
      temp-result)
     (#T
      (send result copy-from temp-result)
      (send result display-yourself))))
   (#T
    (list 'cannot 'fast-intersect 'unlike 'relations)))
))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; RELATION PROJECT
; - build the new record class if necessary
; - instantiate the result-relation if necessary
; - set the record-type of the result-relation to the
;   new record class if necessary
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(relation define-method project (field-list result
                                            new-rec-class)
 (let ((result-relation nil)
       (temp-rec nil)
       (my-members nil))
  (if (or (null? new-rec-class) (symbol? new-rec-class))
   (begin
    (if (null? new-rec-class)
        (set! new-rec-class (gensym)))
    (let ((class-variables (send record-type
                                 get-class-variables))
          (new-class-variables nil)
          (instance-variables (send record-type
                                    get-instance-variables))
          (new-instance-variables nil)
          (superclasses
            (send record-type get-superclasses))
          (new-superclasses nil))
     (while (not (null? class-variables))
      (if (member (caar class-variables) field-list)
       (set! new-class-variables
         (append new-class-variables
                 (car class-variables))))
      (set! class-variables (cdr class-variables)))
     (while (not (null? instance-variables))
      (if (member (caar instance-variables) field-list)
       (set! new-instance-variables
         (append new-instance-variables
                 (car instance-variables))))
      (set! instance-variables
        (cdr instance-variables)))
     (while (not (null? superclasses))
      (if (member (car superclasses) field-list)
       (set! new-superclasses
         (append new-superclasses
                 (car superclasses))))
      (set! superclasses (cdr superclasses)))
     (define-class new-rec-class
       (append '(class-variables) new-class-variables)
       (append '(instance-variables)
               new-instance-variables)
       (append '(superclasses) new-superclasses)
  ))
; new-rec-type class has now been created
```

```
  (cond
   ((null? result)
    (set! result-relation (gensym))
    (send relation make-instance result-relation))
   ((symbol? result)
    (set! result-relation result)
    (send relation make-instance result-relation))
   (#T
    (set! result-relation result)))
  (if (not (equal? (send result-relation
                         get-record-type)
                   new-rec-class))
   (send result-relation set-record-type new-rec-class))
; result-relation has now been created,
;  with its record-type set to the new-rec-class
  (set! my-members members)
  (while (not (null? my-members))
   (set! temp-rec (gensym))
   (send new-rec-class make-instance (eval temp-rec))
   (send (eval temp-rec) copy-from (car my-members))
   (send result-relation add-member temp-rec)
   (set! my-members (cdr my-members))
; result relation is now the result of the project
  (cond
   ((null? result)
    (send result-relation display-yourself))
   ((symbol? result)
    result-relation)
   (#T
    (set! result result-relation)
    (send result display-yourself)
))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; RELATION SELECT
; - choose records that meet the selection criteria
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(relation define-method select (which op what result)
 (let ((my-members members)
       (result-members nil)
       (temp-result nil))
  (cond
   ((or (null? result)
        (symbol? result)
        (equal? (send result class-of-object?)
                'relation))
    (while (not (null? my-members))
      (if (send (eval (car my-members))
            meet-select-criteria? which op what)
        (set! result-members
          (append result-members
                  (list (car my-members)))))
      (set! my-members (cdr my-members))
    )
    (send relation make-instance temp-result)
    (send temp-result set-members result-members)
    (cond
     ((null? result)
      (send temp-result display-yourself))
     ((symbol? result)
      temp-result)
     (#T
      (send result copy-from temp-result)
      (send result display-yourself)))
   (#T
    (list 'invalid 'result)))
))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; RELATION UNION
; - choose every record that is a member of either self
;   or other, using EQUAL-TO? to eliminate duplicates
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(relation define-method union (other result)
 (let ((add-flag #T)    (my-members members)
       (other-members (send other get-members))
       (shared-result-members members)
       (result-members nil)
       (temp-result nil))
  (cond
   ((and (equal? (send other class-of-object?)
                 'relation)
         (or (null? result)
             (symbol? result)
             (equal? (send result class-of-object?)
                     'relation)))
    (while (not (null? other-members))
      (while (not (null? my-members))
        (cond
         ((send (eval (car my-members)) equal-to?
                (eval (car other-members)))
          (set! my-members nil)
          (set! add-flag nil))
         (#T
          (set! my-members (cdr my-members)))
        ))
      (if add-flag
        (set! shared-result-members
          (append shared-result-members
                  (list (car other-members))))
        (set! add-flag #T))
      (set! my-members members)
      (set! other-members (cdr other-members))
    )
    (send relation make-instance temp-result)
    (temp-result set-members shared-result-members)
    (cond
     ((null? result)
      (send temp-result display-yourself))
     ((symbol? result)
      temp-result)
     (#T
      (send result copy-from temp-result)
      (send result display-yourself))))
   (#T
    (list 'cannot 'union 'unlike 'relations))
  )
))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; RECORD
; - Description
;   - A generic-class, intended to be instantiated for
;     every user-defined record class
; - Superclasses
;   - none
; - Class Variables
;   - none
; - Instance Variables
;   - none
; - Methods
;   - generic methods
;     - COPY-FROM
;     - DISPLAY-YOURSELF
;     - EQUAL-TO?
;   - (send x MEET-SELECT-CRITERIA? which op what)
;     - used by the SELECT method of RELATION
;     - which is the instance variable or superclass
;     - op is the operation
;     - what is what to compare the which to
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-generic-class record
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; RECORD INSTANTIATE-GENERIC-METHODS
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(record instantiate-generic-method copy-from (other) ())

(record instantiate-generic-method display-yourself
 () () )

(record instantiate-generic-method equal-to? (other) ())
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; RECORD MEET-SELECT-CRITERIA?
; - check truth of (send which op what)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(record define-method meet-select-criteria?
        (which op what)
 (let ((my-class (send self class-of-object?))
       (my-instvars
         (send my-class get-instance-variables))
       (my-superclasses
         (send my-class get-superclasses))
       (return-value #F))
  (cond
   ((embedded-member? which my-instvars)
    (if (eval '(,op ,which ,what))
      (set! return-value #T)))
   ((embedded-member? which my-superclasses)
    (if (eval '(send ,(eval which) ,op ,what))
      (set! return-value #T))))
  return-value
))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ROOMS-10.EOO - section 10 of ROOMS in EOOPS
; - this section contains the descriptions for the
;   database user's records
;   - EMP-REC
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; EMP-REC
; - Description
;   - Employee Record's
; - Superclasses
;   - RECORD (an instantiation of the generic-class)
;   - PERSON-NAME
;   - ADDR
;   - WIDGET
;   - SALARY
; - Class Variables
;   - none
; - Instance Variables
;   - none
; - Methods
;   - all methods defined for the generic-class record
;   - none defined locally
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-class emp-rec
  (instantiate-generic-class record)
  (superclasses person-name addr widget salary)
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ROOMS-20.EOO - section 20 of ROOMS in EOOPS
; - this section contains the descriptions for all
;   user-defined fields
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; NAME
; - Description
;   - contains a single value (atom, string, list, etc)
;     which is the "name" of something
; - Superclasses
;   - none
; - Class Variables
;   - none
; - Instance Variables
;   - VALUE
; - Methods
;   - generic methods
;     - COPY-FROM
;     - DISPLAY-YOURSELF
;     - EQUAL-TO?
;   - (send x CHANGE-TO new-name)
;     - change value to new-name
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-class name
  (instance-variables (value nil)
   options gettable settable)
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; NAME INSTANTIATE-GENERIC-METHODS
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(name instantiate-generic-method copy-from (other) ())

(name instantiate-generic-method display-yourself () ()
)

(name instantiate-generic-method equal-to? (other) ())
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; NAME CHANGE-TO
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(name define-method change-to (new-name)
  (set! value new-name)
  new-name
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; LAST-NAME
; - Description
;   - contains a last name
; - Superclasses
;   - NAME
; - Class Variables
;   - none
; - Instance Variables
;   - none
; - Methods
;   - none
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-class last-name
  (superclasses name)
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; FIRST-NAME
; - Description
;   - contains a first name
; - Superclasses
;   - NAME
; - Class Variables
;   - none
; - Instance Variables
;   - none
; - Methods
;   - none
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-class first-name
  (superclasses name)
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; MIDDLE-INITIAL
; - Description
;   - contains a middle initial
; - Superclasses
;   - NAME
; - Class Variables
;   - none
; - Instance Variables
;   - none
; - Methods
;   - none
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-class middle-initial
  (superclasses name)
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; PERSON-NAME
; - Description
;   - contains a persons name
;     (last, first, middle initial)
; - Superclasses
;   - LAST-NAME
;   - FIRST-NAME
;   - MIDDLE-INITIAL
; - Class Variables
;   - none
; - Instance Variables
;   - none
; - Methods
;   - generic methods
;     - COPY-FROM
;     - DISPLAY-YOURSELF
;     - EQUAL-TO?
;   - (send x CHANGE-LAST-NAME-TO new-last-name)
;     - calls (send last-name change-to new-last-name)
;   - (send x CHANGE-FIRST-NAME-TO new-first-name)
;     - calls (send first-name change-to new-first-name)
;   - (send x CHANGE-MIDDLE-INITIAL-TO new-mi)
;     - calls (send middle-initial change-to new-mi)
;   - (send x CHANGE-TO new-name)
;     - calls
;       (send self change-last-name-to (car new-name))
;       (send self change-first-name-to (cadr new-name))
;       (send self change-middle-initial-to
;             (caddr new-name))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-class person-name
  (superclasses last-name first-name middle-initial)
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; PERSON-NAME INSTANTIATE-GENERIC-METHODS
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(person-name instantiate-generic-method copy-from
  (other) ())

(person-name instantiate-generic-method display-yourself
 () () )

(person-name instantiate-generic-method equal-to?
  (other) ())
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; PERSON-NAME CHANGE-LAST-NAME-TO
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(person-name define-method change-last-name-to
             (new-last-name)
  (send self last-name.change-to new-last-name)
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; PERSON-NAME CHANGE-FIRST-NAME-TO
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(person-name define-method change-first-name-to
             (new-first-name)
  (send self first-name.change-to new-first-name)
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; PERSON-NAME CHANGE-MIDDLE-INITIAL-TO
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(person-name define-method change-middle-initial-to
             (new-mi)
  (send self middle-initial.change-to new-mi)
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; PERSON-NAME CHANGE-TO
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(person-name define-method change-to (new-name)
  (if (and (equal? '3 (length new-name))
           (atom? (car new-name))
           (atom? (cadr new-name))
           (atom? (caddr new-name)))
    (append
      (list (send self change-last-name-to
                  (car new-name)))
      (list (send self change-first-name-to
                  (cadr new-name)))
      (list (send self change-middle-initial-to
                  (caddr new-name))))
    nil)
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ADDR
; - Description
;   - contains an address
;     (street, city, state, zip)
; - Superclasses
;   - NAME, NAME, NAME, NAME
; - Class Variables
;   - none
; - Instance Variables
;   - none
; - Methods
;   - generic methods
;     - COPY-FROM
;     - DISPLAY-YOURSELF
;     - EQUAL-TO?
;   - (send x CHANGE-STREET-TO new-street)
;     - calls (send self name.1.change-to new-street)
;   - (send x CHANGE-CITY-TO new-city)
;     - calls (send self name.2.change-to new-city)
;   - (send x CHANGE-STATE-TO new-state)
;     - calls (send self name.3.change-to new-state)
;   - (send x CHANGE-ZIP-TO new-zip)
;     - calls (send self name.4.change-to new-zip)
;   - (send x CHANGE-TO new-addr)
;     - calls
;       (send self change-street-to (car new-addr))
;       (send self change-city-to (cadr new-addr))
;       (send self change-state-to (caddr new-addr))
;       (send self change-zip-to (cadddr new-addr))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-class addr
  (superclasses name name name name)
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ADDR INSTANTIATE-GENERIC-METHODS
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(addr instantiate-generic-method copy-from (other) ())

(addr instantiate-generic-method display-yourself () ()
)

(addr instantiate-generic-method equal-to? (other) ())
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ADDR CHANGE-STREET-TO
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(addr define-method change-street-to (new-street)
  (send self name.1.change-to new-street)
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ADDR CHANGE-CITY-TO
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(addr define-method change-city-to (new-city)
  (send self name.2.change-to new-city)
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ADDR CHANGE-STATE-TO
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(addr define-method change-state-to (new-state)
  (send name.3.change-to new-state)
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ADDR CHANGE-ZIP-TO
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(addr define-method change-zip-to (new-zip)
  (send self name.4.change-to new-zip)
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ADDR CHANGE-TO
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(addr define-method change-to (new-addr)
  (if (and (equal? '4 (length new-addr))
           (atom? (car new-addr))
           (atom? (cadr new-addr))
           (atom? (caddr new-addr))
           (atom? (cadddr new-addr)))
    (append
      (list (send self change-street-to (car new-addr)))
      (list (send self change-city-to (cadr new-addr)))
      (list (send self change-state-to
                       (caddr new-addr)))
      (list (send self change-zip-to
                       (cadddr new-addr))))
    nil)
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; WIDGET
; - Description
;   - looks amazingly like a telephone number,
;     - x-coord == area code
;     - y-coord == exchange
;     - z-coord == extension
;   - note: does not use generic methods (by choice...)
; - Superclasses
;   - none
; - Class Variables
;   - none
; - Instance Variables
;   - x-coord, y-coord, & z-coord
; - Methods
;   - (send x COPY-FROM other)
;     - if (other is of type widget)
;       set instvars of x to the instvars of other
;   - (send x DISPLAY-YOURSELF)
;     - displays x-coord, y-coord, & z-coord
;   - (send x EQUAL-TO? y)
;     - if (y is of type widget) and
;          (the instance variables of x are equal to the
;           instance variables of y)
;       return #T, otherwise return #F
;   - (send x SET-WIDGET coords)
;     - sets x-coord to (car coords),
;       y-coord to (cadr coords), &
;       z-coord to (caddr coords)
;   - (send x CHANGE-TO new-widget)
;     - calls (send self set-widget new-widget)
;   - (send x CHANGE-X-TO new-x)
;     - sets x-coord to new-x
;   - (send x CHANGE-Y-TO new-y)
;     - sets y-coord to new-y
;   - (send x CHANGE-Z-TO new-z)
;     - sets z-coord to new-z
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-class widget
  (instvars  (x-coord nil) (y-coord nil) (z-coord nil)
   options gettable settable)
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; WIDGET COPY-FROM
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(widget define-method (widget copy-from) (other)
  (cond
   ((equal? (send other class-of-object?) widget)
    (set! x-coord (send other get-x-coord))
    (set! y-coord (send other get-y-coord))
    (set! z-coord (send other get-z-coord))
    self)
   (#T
    nil))
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; WIDGET DISPLAY-YOURSELF
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(widget define-method display-yourself ()
  (append (list x-coord) (list y-coord) (list z-coord))
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; WIDGET EQUAL-TO?
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(widget define-method equal-to? (other)
  (if (and (equal? (send other class-of-object?) widget)
           (equal? x-coord (send other get-x-coord))
           (equal? y-coord (send other get-y-coord))
           (equal? z-coord (send other get-z-coord)))
    #T
    #F)
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; WIDGET SET-WIDGET
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(widget define-method set-widget (coords)
  (if (and (equal? (length coords) '3)
           (atom? (car coords))
           (atom? (cadr coords))
           (atom? (caddr coords)))
    (append (list (set! x-coord (car coords)))
            (list (set! y-coord (cadr coords)))
            (list (set! z-coord (caddr coords))))
    nil)
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; WIDGET CHANGE-TO
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(widget define-method change-to (coords)
  (send self set-widget coords)
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; WIDGET CHANGE-X-TO
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(widget define-method change-x-to (new-x)
  (set! x-coord new-x)
  new-x
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; WIDGET CHANGE-Y-TO
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-method (widget change-y-to) (new-y)
  (set! y-coord new-y)
  new-y
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; WIDGET CHANGE-Z-TO
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(widget define-method change-z-to (new-z)
  (set! z-coord new-z)
  new-z
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; SALARY
; - Description
;   - integer field
;   - note: does not use generic methods (by choice...)
; - Superclasses
;   - none
; - Class Variables
;   - none
; - Instance Variables
;   - VALUE
; - Methods
;   - (send x COPY-FROM other)
;     - if (other is of type salary)
;       set value of x to the value of other
;   - (send x DISPLAY-YOURSELF)
;     - displays value
;   - (send x EQUAL-TO? y)
;     - if (y is of type salary) and
;          (the value of x is equal to the value of y)
;       return #T, otherwise return #F
;   - (send x SET-SALARY new-value)
;     - sets value to new-value
;   - (send x CHANGE-TO new-value)
;     - calls (send self set-salary new-value)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-class salary
  (instance-variables  (value 0)
   options gettable settable)
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; SALARY COPY-FROM
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(salary define-method copy-from (other)
  (cond
   ((equal? (send other class-of-object?) salary)
    (set! value (send other get-value))
    self)
   (#T
    nil))
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; SALARY DISPLAY-YOURSELF
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(salary define-method display-yourself ()
  (list value)
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; SALARY EQUAL-TO?
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(salary define-method equal-to? (other)
  (if (and (equal? (send other class-of-object?) salary)
           (equal? value (send other get-value)))
    #T
    #F)
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; SALARY SET-SALARY
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(salary define-method set-salary (new-value)
  (if (not (number? new-value))
    nil
    (set! value new-value))
)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; SALARY CHANGE-TO
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(salary define-method change-to (new-value)
  (send self set-salary new-value)
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; UTILS.EOO - utilities used by ROOMS
; - this section contains the descriptions for
;   - (concat x y)
;     - concatenates x & y together
;   - (embedded-member? x y)
;     - determines if x is contained anywhere within y
;   - (list? x)
;     - determines if x is a list
;   - (pprint x)
;     - prints each member of x on a separate line
;   - (subset? x y)
;     - determines if x is a subset of y
;   - (while condition statements)
;     - while condition is true, executes statements
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; (CONCAT x y)
; - concatenates x and y together,
;   - x and y must be atoms
; - ex, (concat 'a 'b)    = 'ab
;       (concat 'ab 'xyz) = 'abxyz
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define concat (lambda (x y)
  (implode (append (explode x) (explode y)))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; (EMBEDDED-MEMBER? x y)
; - determines if x appears anywhere in y
;   - x can be an atom or a list, y must be a list
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (embedded-member? x y)
 (let ((return-value #F)
       (finished #F))
  (if (list? y)
    (while (not finished)
      (if (or (member x y)
              (and (list? (car y))
                   (embedded-member? x (car y))))
        (begin
         (set! return-value #T)
         (set! finished #T))
        (begin
          (set! y (cdr y))
          (if (null? y) (set! finished #T))))
    ))
  return-value
))



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; (LIST? x)
; - determines if x is a list
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(macro list? (lambda (e) '(not (atom? ,(cadr e)))))



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; (PPRINT x)
; - display each member of the list x on a separate line
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (pprint x)
  (while (>? (length x) 1)
    (print (car x))
    (set! x (cdr x))
  )
  (newline)
  (car x)
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; (SUBSET? l1 l2)
; - returns #T if l1 is a subset of l2
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (subset? l1 l2)
 (cond
  ((null? l1)                #T)
  ((member (car l1) l2)      (subset? (cdr l1) l2))
  (t                         #F)
))



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; (WHILE condition statements)
; - while condition is #T, execute statements
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(macro while
  (lambda (e)
    '(do #!null ((not,(cadr e))),@(cddr e))
))
```

BIBLIOGRAPHY

Andrews, T., and Harris, C. "Combining Language and Database Advances in an Object-Oriented Development Environment." OOPSLA'87 Proceedings, Oct. 1987, Orlando, FL; Special Issue of SIGPLAN Notices, Vol. 22, No. 12, Dec. 1987, pp. 430-440.

Banerjee, J.; Kim, W.; Kim, H-J., and Korth, H. F. "Semantics and Implementation of Schema Evolution in Object-Oriented Databases." Proceedings of ACM SIGMOD'87 Annual Conference, San Francisco, CA, May 1987; SIGMOD Record, Vol. 16, No. 3, Dec. 1987, pp. 311-322.

Beech, D. "Intensional Concepts in an Object Database Model." OOPSLA'88 Proceedings, Sept. 1988, San Diego, CA; Special Issue of SIGPLAN Notices, Vol. 23, No. 11, Nov. 1988, pp. 164-175.

Bic, L., and Gilbert, J. P. "Learning from AI: New Trends in Database Technology." IEEE Computer, Vol. 19, No. 3, Mar. 1986, pp. 44-54.

Blaha, M. R.; Premerlani, W. J., and Rumbaugh, J. E. "Relational Database Design Using an Object-Oriented Methodology." Communications of the ACM, Vol. 31, No. 4, Apr. 1988, pp. 414-427.

Bloom, T., and Zdonik, S. B. "Issues in the Design of Object-Oriented Database Programming Languages." OOPSLA'87 Proceedings, Oct. 1987, Orlando, FL; Special Issue of SIGPLAN Notices, Vol. 22, No. 12, Dec. 1987, pp. 441-451.

Bobrow, D. G.; Kahn, K.; Kiczales, G.; Masinter, L.; Stefik, M., and Zdybel, F. "CommonLoops: Merging Lisp and Object-Oriented Programming." OOPSLA'86 Proceedings, Sept.-Oct. 1986, Portland, OR; Special Issue of SIGPLAN Notices, Vol. 21, No. 11, Nov. 1986, pp. 17-29.

Bobrow, D. G.; DeMichiel, L. G.; Gabriel, R. P.; Keene, S. E.; Kiczales, G., and Moon, D. A. "Common Lisp Object System Specification X3J13 Document 88-002R." Special Issue of SIGPLAN Notices, Vol. 23, No. 9, Sept. 1988, pp. 1-1 - 2-94.

Booch, G. "Object-Oriented Development." IEEE Transactions on Software Engineering, Vol. SE-12, No. 2, Feb. 1986, pp. 211-221.

Booch, G. Software Engineering with Ada. Menlo Park, CA: Benjamin/Cummings Publishing Co., Inc., 1987.

Borning, A. H., and Ingalls, D. H. "Multiple Inheritance in Smalltalk-80." Proceedings of the National Conference on Artificial Intelligence (AAAI-82), Pittsburgh, PA, Aug. 1982, pp. 234-237.

Brackett, M. H. Developing Data Structured Databases. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1987.

Briot, J-P., and Cointe, P. "A Uniform Model for Object-Oriented Languages Using the Class Abstraction." Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI87), Milan, Italy, Aug. 1987, pp. 40-43.

Buneman, P., and Atkinson, M. "Inheritance and Persistence in Database Programming Languages." Proceedings of ACM SIGMOD '86 International Conference on Management of Data, Washington, D.C., May 1986; SIGMOD Record, Vol. 15, No. 2, June 1986, pp. 4-15.

Cardelli, L., and Wegner, P. "On Understanding Types, Data Abstraction, and Polymorphism." Computing Surveys, Vol. 17, No. 4, Dec. 1985, pp. 471-522.

Cardelli, L.; Donahue, J.; Glassman, L.; Jordan, M.; Kalsow, B., and Nelson, G. Modula-3 Report. Palo Alto, CA: Digital Equipment Corporation Systems Research Center, 1988.

Caruso, M.; Strong, R.; Williams, M.; Zdonik, S., and Nastos, N. Object-Oriented Database Systems. OOPSLA'87 Tutorial, Oct. 1987, Orlando, FL.

Caruso, M. J. The Evolution and Design of an Object-Oriented Database Management System. OOPSLA'88 Tutorial, Sept. 1988, San Diego, CA.

Christodoulakis, S.; Ho, F., and Theodoridou, M. "The Multimedia Object Presentation Manager of MINOS: A Symmetric Approach." Proceedings of ACM SIGMOD'86 International Conference on Management of Data, Washington, D.C., May 1986; SIGMOD Record, Vol. 15, No. 2, June 1986, pp. 295-310.

Codd, E. F. "A Relational Model of Data for Large Shared Data Banks." Communications of the ACM, Vol. 13, No. 6, June 1970, pp. 377-387.

Cointe, P. "Metaclasses are First Class: the ObjVlisp Model." OOPSLA'87 Proceedings, Oct. 1987, Orlando, FL; Special Issue of SIGPLAN Notices, Vol. 22, No. 12, Dec. 1987, pp. 156-167.

Conrad, R.; Piersol, K., and Van Orden, E. The Analyst Project: A Smalltalk-80 Application Case Study. OOPSLA'87 Tutorial, Oct. 1987, Orlando, FL.

Copeland, G., and Maier, D. "Making Smalltalk a Database System." Proceedings of the ACM SIGMOD, Boston, MA, June 1984, pp. 316-325.

Cox, B. J. "Message/Object Programming: An Evolutionary Change in Programming Technology." IEEE Software, Vol. 1, No. 1, Jan. 1984, pp. 50-61.

Cox, B. J. Object-Oriented Programming: An Evolutionary Approach. Reading, MA: Addison-Wesley Publishing Co., 1986.

Danforth, S., and Tomlinson, C. "Type Theories and Object-Oriented Programming." Computing Surveys, Vol. 20, No. 1, Mar. 1988, pp. 29-72.

Diederich, J., and Milton, J. "Experimental Prototyping in Smalltalk." IEEE Software, Vol. 4, No. 3, May 1987, pp. 50-64.

Digitalk Inc. Smalltalk/V: Tutorial and Programming Handbook. Los Angeles: Digitalk Inc., 1986.

Duhl, J., and Damon, C. "A Performance Comparison of Object and Relational Databases Using the Sun Benchmark." OOPSLA'88 Proceedings, Sept. 1988, San Diego, CA; Special Issue of SIGPLAN Notices, Vol. 23, No. 11, Nov. 1988, pp. 153-163.

Fishman, D. H.; Beech, D.; Cate, H. P.; Chow, E. C.; Connors, T.; Davis, J. W.; Derrett, N.; Hoch, C. G.; Kent, W.; Lyngbaek, P.; Mahbod, B.; Neimat, M. A.; Ryan, T. A., and Shan, M. C. "Iris: An Object-Oriented Database Management System." ACM Transactions on Office Information Systems, Vol. 5, No. 1, Jan. 1987, pp. 48-69.

Gallaire, H.; Minker, J., and Nicolas, J-M. "Logic and Databases: A Deductive Approach." Computing Surveys, Vol. 16, No. 2, June 1984, pp. 153-185.

Graphael. The Data and Knowledge Management Company. Waltham, MA: Graphael, 1988.

Halbert, D. C., and O'Brien, P. D. "Using Types and Inheritance in Object-Oriented Programming." IEEE Software, Vol. 4, No. 5, Sept. 1987, pp. 71-79.

Hendler, J. "Enhancement for Multiple-Inheritance." SIGPLAN Notices, Vol. 21, No. 10, Oct. 1986, pp. 98-106.

Hudson, S. E., and King, R. "Object-Oriented Database Support for Software Environments." Proceedings of ACM SIGMOD'87 Annual Conference, San Francisco, CA, May 1987; SIGMOD Record, Vol. 16, No. 3, Dec. 1987, pp. 491-503.

Hutt, A. E. "Data Base Management and Administration." In Computer Handbook for Senior Management, pp. 119-135. Edited by D. B. Hoyt. New York: MacMillan Publishing Co, Inc., 1978.

Jacky, J. P., and Kalet, I. J. "An Object-Oriented Programming Discipline for Standard Pascal." Communications of the ACM, Vol. 30, No. 9, Sept. 1987, pp. 772-776.

Kemper, A.; Lockemann, P. C., and Wallrath, M. "An Object-Oriented Database System for Engineering Applications." Proceedings of ACM SIGMOD'87 Annual Conference, San Francisco, CA, May 1987; SIGMOD Record, Vol. 16, No. 3, Dec. 1987, pp. 299-310.

Kernighan, B. W., and Ritchie, D. M. The C Programming Language. Engelwood Cliffs, NJ: Prentice-Hall, Inc., 1978.

Khoshafian, S. N., and Copeland, G. P. "Object Identity." OOPSLA'86 Proceedings, Sept.-Oct. 1986, Portland, OR; Special Issue of SIGPLAN Notices, Vol. 21, No. 11, Nov. 1986, pp. 406-416.

Kim, W.; Banerjee, J.; Chou, H-T.; Garza, J. F., and Woelk, D. "Composite Object Support in an Object-Oriented Database System." OOPSLA'87 Proceedings, Oct. 1987, Orlando, FL; Special Issue of SIGPLAN Notices, Vol. 22, No. 12, Dec. 1987, pp. 118-125.

Kim, W.; Ballou, N.; Banerjee, J.; Chou, H-T.; Garza, J. F., and Woelk, D. "Integrating an Object-Oriented Programming System with a Database System." OOPSLA'88 Proceedings, Sept. 1988, San Diego, CA; Special Issue of SIGPLAN Notices, Vol. 23, No. 11, Nov. 1988, pp. 142-152.

Lieberman, H. "Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems." OOPSLA'86 Proceedings, Sept.-Oct. 1986, Portland, OR; Special Issue of SIGPLAN Notices, Vol. 21, No. 11, Nov. 1986, pp. 214-223.

Maier, D.; Stein, J.; Otis, A., and Purdy, A. "Development of an Object-Oriented DBMS." OOPSLA'86 Proceedings, Sept.-Oct. 1986, Portland, OR; Special Issue of SIGPLAN Notices, Vol. 21, No. 11, Nov. 1986, pp. 472-482.

Martin, J. Principles of Data-Base Management. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1976.

McGregor, J. D. "Object-Oriented Programming with SCOOPS." Computer Language, Vol. 4, No. 7, July 1987, pp. 49-56.

Merrow, T., and Laursen, J. "A Pragmatic System for Shared Persistent Objects." OOPSLA'87 Proceedings, Oct. 1987, Orlando, FL; Special Issue of SIGPLAN Notices, Vol. 22, No. 12, Dec. 1987, pp. 103-110.

Meyer, B. "Genericity versus Inheritance." OOPSLA'86 Proceedings, Sept.-Oct. 1986 Portland, OR; Special Issue of SIGPLAN Notices, Vol. 21, No. 11, Nov. 1986, pp. 391-405.

Meyer, B. "Eiffel: Programming for Reusability and Extendibility." SIGPLAN Notices, Vol. 22, No. 2, Feb. 1987a, pp. 85-94.

Meyer, B. "Reusability: The Case for Object-Oriented Design." IEEE Software, Vol. 4, No. 2, Mar. 1987b, pp. 50-64.

Meyer, B. Eiffel: An Introduction. Goleta, CA: Interactive Software Engineering Inc., 1988a.

Meyer, B. Object-Oriented Design for Software Engineering. OOPSLA'88 Tutorial, Sept. 1988b, San Diego, CA.

Milani, M., and Ege, R. An Introduction to Object-Oriented Concepts. OOPSLA'88 Tutorial, Sept. 1988, San Diego, CA.

Moon, D. A. "Object-Oriented Programming with FLAVORS." OOPSLA'86 Proceedings, Sept.-Oct. 1986, Portland, OR; Special Issue of SIGPLAN Notices, Vol. 21, No. 11, Nov. 1986, pp. 1-8.

Ong, J.; Fogg, D., and Stonebraker, M. "Implementation of Data Abstraction in the Relational Database System INGRES." SIGMOD Record, Vol. 14, No. 1, Mar. 1984, pp. 1-14.

Ontologic Inc. Vbase: For Object Applications. Billerica, MA: Ontologic Inc., 1988.

Orenstein, J. A. "Spatial Query Processing in an Object-Oriented Database System." Proceedings of ACM SIGMOD '86 International Conference on Management of Data, Washington, D.C., May 1986; SIGMOD Record, Vol. 15, No. 2, June 1986, pp. 326-336.

Osborn, S. L., and Heaven, T. E. "The Design of a Relational Database System with Abstract Data Types for Domains." ACM Transactions on Database Systems, Vol. 11, No. 3, Sept. 1986, pp. 357-373.

Papazoglou, M. P.; Georgiadis, P. I., and Maritsas, D. G. "Object-Oriented Programming With SCOOPS." Computer Languages, Vol. 9, No. 2, 1984, pp. 107-131.

Penney, D. J., and Stein, J. "Class Modification in the GemStone Object-Oriented DBMS." OOPSLA'87 Proceedings, Oct. 1987, Orlando, FL; Special Issue of SIGPLAN Notices, Vol. 22, No. 12, Dec. 1987, pp. 111-117.

Pinson, L. J, and Wiener, R. S. An Introduction to Object-Oriented Programming and Smalltalk. Reading: Addison-Wesley Publishing Co., 1988.

Reiter, R. "On Closed World Data Bases." In Logic and Data Bases. Edited by H. Gallaire and J. Minker. New York: Plenum Press, 1978.

Rentsch, T. "Object Oriented Programming." SIGPLAN Notices, Vol. 17, No. 9, Sept. 1982, pp. 51-57.

Rumbaugh, J. "Relations as Semantic Constructs in an Object-Oriented Language." OOPSLA'87 Proceedings, Oct. 1987, Orlando, FL; Special Issue of SIGPLAN Notices, Vol. 22, No. 12, Dec. 1987, pp. 466-481.

Sandberg, D. "An Alternative to Subclassing." OOPSLA'86 Proceedings, Sept.-Oct. 1986, Portland, OR; Special Issue of SIGPLAN Notices, Vol. 21, No. 11, Nov. 1986, pp. 424-428.

Schaffert, C.; Cooper, T.; Bullis, B.; Kilian, M., and Wilpolt, C. "An Introduction to Trellis/Owl." OOPSLA'86 Proceedings, Sept.-Oct. 1986, Portland, OR; Special Issue of SIGPLAN Notices, Vol. 21, No. 11, Nov. 1986, pp. 9-16.

Schmucker, K. An Introduction To Object-Oriented Concepts. OOPSLA'87 Tutorial, Oct. 1987, Orlando, FL.

Seidewitz, E. "Object-Oriented Programming in Smalltalk and Ada." OOPSLA'87 Proceedings, Oct. 1987, Orlando, FL; Special Issue of SIGPLAN Notices, Vol. 22, No. 12, Dec. 1987, pp. 202-213.

Servio Logic Corp. GemStone: Product Overview. Alameda, CA: Servio Logic Corp., 1988a.

Servio Logic Corp. Selecting an Object-Oriented Database Management System. Alameda, CA: Servio Logic Corp., 1988b.

Shaw, M. "Abstraction Techniques in Modern Programming Languages." IEEE Software, Vol. 1, No. 4, Oct. 1984, pp. 10-26.

Skarra, A. H., and Zdonik, S. B. "The Management of Changing Types in an Object-Oriented Database." OOPSLA'86 Proceedings, Sept.-Oct. 1986, Portland, OR; Special Issue of SIGPLAN Notices, Vol. 21, No. 11, Nov. 1986, pp. 483-495.

Sloan, A.; Carolan, J., and Dockrell, A. C++ and Object-Oriented Design. OOPSLA'88 Tutorial, Sept. 1988, San Diego, CA.

Smith, K. E., and Zdonik, S. B. "Intermedia: A Case Study of the Differences Between Relational and Object-Oriented Database Systems." OOPSLA'87 Proceedings, Oct. 1987, Orlando, FL; Special Issue of SIGPLAN Notices, Vol. 22, No. 12, Dec. 1987, pp. 452-465.

Snyder, A. "Common Objects: An Overview." SIGPLAN Notices, Vol. 21, No. 10, Oct. 1986a, pp. 19-28.

Snyder, A. "Encapsulation and Inheritance in Object-Oriented Programming Languages." OOPSLA'86 Proceedings, Sept.-Oct. 1986, Portland, OR; Special Issue of SIGPLAN Notices, Vol. 21, No. 11, Nov. 1986b, pp. 38-45.

Stefik, M., and Bobrow, D. G. "Object-Oriented Programming: Themes and Variations." AI Magazine, Vol. 6, No. 4, Winter 1986, pp. 40-62.

Stein, L. A. "Delegation is Inheritance." OOPSLA'87 Proceedings, Oct. 1987, Orlando, FL; Special Issue of SIGPLAN Notices, Vol. 22, No. 12, Dec. 1987, pp. 138-146.

Stonebraker, M., and Rowe, L. A. "The Design of POSTGRES." SIGMOD Record, Vol. 15, No. 2, June 1986, pp. 340-355.

Stroustrup, B. "An Overview of C++." SIGPLAN Notices, Vol. 21, No. 10, Oct. 1986a, pp. 7-18.

Stroustrup, B. The C++ Programming Language. Reading, MA: Addison-Wesley Publishing Co., 1986b.

Stroustrup, B. "What is Object-Oriented Programming?" IEEE Software, Vol. 5, No. 3, May 1988, pp. 10-20.

Texas Instruments Inc. PC Scheme: User's Guide. Revision B. Austin, TX: Texas Instruments Inc., 1987a.

Texas Instruments Inc. TI Scheme: Language Reference Manual. Revision B. Austin, TX: Texas Instruments Inc., 1987b.

Thatte, S. T. Report on the Object-Oriented Database Workshop: Implementation Aspects. Held in conjunction with OOPSLA'87, Oct. 1987, Orlando, FL.

Thomas, D. Survey of Object-Oriented Programming Systems. OOPSLA'87 Tutorial, Oct. 1987, Orlando, FL.

Touati, H. "Is Ada an Object Oriented Programming Language?" SIGPLAN Notices, Vol. 22, No. 5, May 1987, pp. 23-26.

Ullman, J. D. Principles of Database Systems. Rockville, MD: Computer Science Press, 1982.

Ungar, D., and Smith, R. B. "Self: The Power of Simplicity." OOPSLA'87 Proceedings, Oct. 1987, Orlando, FL; Special Issue of SIGPLAN Notices, Vol. 22, No. 12, Dec. 1987, pp. 227-242.

Wegner, P. "Classification in Object-Oriented Systems." SIGPLAN Notices, Vol. 21, No. 10, Oct. 1986, pp. 173-182.

Wegner, P. "Dimensions of Object Based Language Design." OOPSLA'87 Proceedings, Oct. 1987, Orlando, FL; Special Issue of SIGPLAN Notices, Vol. 22, No. 12, Dec. 1987, pp. 168-182.

Wiederhold, G. "Views, Objects, and Databases." IEEE Computer, Vol. 19, No. 12, Dec. 1986, pp. 37-44.

Wiener, R. S., and Pinson, L. J. An Introduction to Object-Oriented Programming and C++. Reading, MA: Addison-Wesley Publishing Co., 1988.

Woelk, D.; Kim, W., and Luther, W. "An Object-Oriented Approach to Multimedia Databases." Proceedings of ACM SIGMOD'86 International Conference on Management of Data, Washington, D.C., May 1986; SIGMOD Record, Vol. 15, No. 2, June 1986, pp. 311-325.

Xerox. Analyst Product Description. Pasadena, CA: Xerox Special Information Systems, June 1987.

Xerox. The Analyst. Narrated by Kurt Piersol, 30 min., Xerox Special Information Systems, Mar. 1988, videocassette.

Zdonik, S.; Moss, E., and Herlihy, M. Object-Oriented Databases. OOPSLA'88 Tutorial, Sept. 1988, San Diego, CA.

Zortech Inc. Zortech C++ Compiler. Arlington, MA: Zortech Inc., 1988.

UNIVERSITY OF CENTRAL FLORIDA

OFFICE OF GRADUATE STUDIES

DISSERTATION APPROVAL

DATE: November 9, 1988

BASED ON THE CANDIDATE'S SUCCESSFUL ORAL DEFENSE, IT IS RECOMMENDED THAT THE DISSERTATION PREPARED BY Michael L. Nelson

ENTITLED "A Relational Object-Oriented Management System and an Encapsulated Object-Oriented Programming System"

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Doctor of Philosophy

FROM THE DEPARTMENT OF Computer Science

IN THE COLLEGE OF Arts and Sciences

ali orooji

Ali Orooji, Major Professor
Department of Computer Science

Terry J. Frederick

Terry J. Frederick, Graduate Coordinator
Department of Computer Science

Jack B. Rollins

Jack B. Rollins, Dean
College of Arts & Sciences

L. M. Trefonas

Louis M. Trefonas
Dean of Graduate Studies

UNIVERSITY OF CENTRAL FLORIDA

OFFICE OF GRADUATE STUDIES

DEFENSE OF DISSERTATION

THE UNDERSIGNED VERIFY THAT THE FINAL ORAL DEFENSE OF THE DOCTOR OF PHILOSOPHY DISSERTATION OF Michael L. Nelson HAS BEEN SUCCESSFULLY COMPLETED ON November 9, 1988

TITLE OF DISSERTATION: "A Relational Object-Oriented Management System and an Encapsulated Object-Oriented Programming System"

MAJOR FIELD OF STUDY: Computer Science

COMMITTEE:

ali orooji

Chairperson - Ali Orooji

Member - J. Michael Moshell

Member - Charles Hughes

Member - Harley Myler

APPROVED:

12/5/88

L. M. Trefonas Date

Dean of Graduate Studies